RADC-TR-90-404, Vol IV (of 18)
Final Technical Report
December 1990

DISTRIBUTED ARTIFICIAL INTELLIGENCE FOR COMMUNICATIONS NETWORK MANAGEMENT

Northeast Artificial Intelligence Consortium (NAIC)

Robert A. Meyer and Susan E. Conry

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-404, Volume IV (of 18) has been reviewed and is approved for publication.

APPROVED: 

ARLAN L. MORSE, Captain, USAF
Project Engineer

APPROVED: 

JOHN A. GRANIERO
Technical Director
Directorate of Communications

FOR THE COMMANDER: 

BILLY G. OAKS
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (DCLD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final Sep 84 - Dec 89

4. TITLE AND SUBTITLE
DISTRIBUTED ARTIFICIAL INTELLIGENCE FOR COMMUNICATIONS NETWORK MANAGEMENT

5. FUNDING NUMBERS
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

6. AUTHOR(S)
Robert A. Meyer and Susan E. Conry

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-404, Vol IV (of 18)

11. SUPPLEMENTARY NOTES (See reverse)
RADC Project Engineer: Arlan L. Morse, Captain, USAF/DCLD/(315) 330-7751

This effort was funded partially by the Laboratory Director's fund.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the use of knowledge based systems for communications network management and control via an architecture for a diversely distributed multi-agent system.

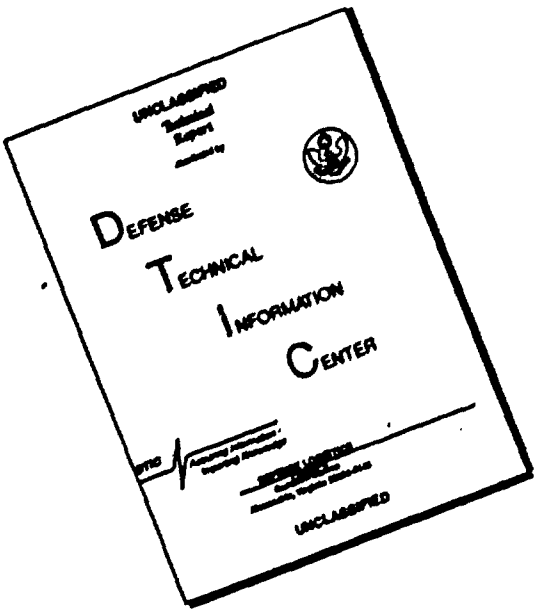
14. SUBJECT TERMS Artificial Intelligence, Distributed Artificial Intelligence, Distributed Planning, Knowledge Based Reasoning, Simulation, Communications Network, Graphical User Interface

15. NUMBER OF PAGES 162

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

DISCLAIMER NOTICE



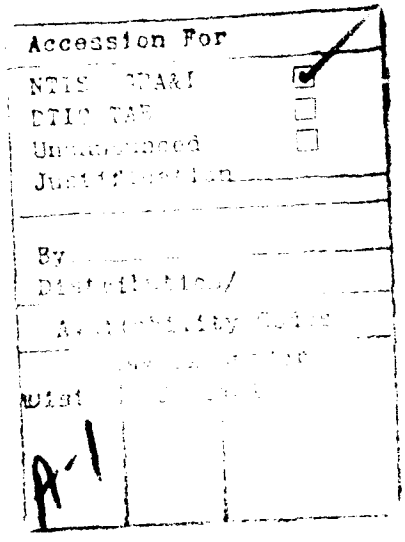
THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Block 5 (Cont'd) Funding Numbers

PE - 62702F	PE - 61102F	PE - 61102F	PE - 33126F	PE - 61101F
PR - 5581	PR - 2304	PR - 2304	PR - 2155	PR - LDFP
TA - 27	TA - J5	TA - J5	TA - 02	TA - 27
WU - 23	WU - 01	WU - 15	WU - 10	WU - 01

Block 11 (Cont'd)

This effort was performed as a subcontract by Clarkson University to Syracuse University, Office of Sponsored Programs.



Contents

List of Figures

List of Tables

4.1 Executive Summary

This report describes the results of a five year research project performed at Clarkson University as part of the Northeast Artificial Intelligence Consortium (NAIC). The major objectives of this research were to gain a better understanding of issues that arise in the area of distributed artificial intelligence (DAI) and to investigate the application of DAI in communications network management and control. This report gives a brief overview of the problem domain and describes the primary results in three areas. First, a model for distributed, intelligent network management has been developed. Second, a distributed AI system testbed (DAISY) was designed and implemented. Third, contributions to DAI have been made in the form of three distinct systems which have been implemented to test our theories and design strategies for DAI systems. The report also documents the building of an AI infrastructure which will support the growth of future AI research at Clarkson, both in conjunction with other universities in the NAIC and with the U.S. Air Force at Rome Air Development Center.

Distributed artificial intelligence is concerned with issues that arise when loosely coupled problem solving agents work collectively to solve a problem. These agents are typically characterized as having a degree of functional specialization, a local perspective, and incomplete knowledge. Although each agent uses its own local perspective and knowledge in performing its tasks, a complete solution to the problem usually requires cooperation among the agents. One of the most important issues in these problems is which cooperation paradigms are most appropriate.

The strategy followed in this research was first to choose an application problem domain to which DAI might be applicable. This domain should have a structure and degree of complexity that make it a realistic one for investigating these problems. The monitoring and control of large communications systems was selected because it is a naturally distributed, complex problem which currently requires cooperation among humans for an effective solution. The problem is an important one to the Department of Defense, and it is rich both in problems to be solved and in structure that can be used to investigate distributed problem solving paradigms.

The model we have used for a communications network is based on the large scale, world-wide Defense Communication System (DCS). We have concentrated on network management and control at the subregion level. The subregion level represents a group of several individual sites or nodes in the communications system architecture which are monitored and controlled from a single control center. System-wide management and control is distributed over a network of subregion control centers, typically from three to twelve in number. Our view of the role of DAI in this environment is to

provide cooperating, intelligent, semi-autonomous agents to serve as problem solving assistants to the human controllers.

We have developed an architecture for this system in which each component (an agent) is a specialized and localized knowledge-based system designed to provide assistance to the human operator. Analysis of the problem solving activities currently employed by these operators found three fundamental kinds of functions required: (1) data interpretation and situation assessment, (2) diagnosis and fault isolation; and (3) planning to find and allocate scarce resources for restoral of service in the event of an outage. This suggested a natural functional distribution of agents. Not only must each agent be able to cooperate with other agents performing different functions at the same local site, but each agent must also be able to cooperate with identical agents located in physically separate facilities. This concept of a knowledge-based network management system as a spatially and functionally distributed collection of semi-autonomous, cooperating intelligent agents is an important contribution to the development of future network management and control systems for the DCS.

At a local level the system is seen as a number of functionally specialized agents that cooperate in a loosely coupled fashion. These agents comprise a local participant in a network-wide team of problem solvers. At the global level, the system may be viewed as a group of relatively independent, spatially distributed problem solving systems cooperating to solve a collection of problems. A key characteristic of this model is the representation of knowledge in a distributed manner. Our results include the development of a distributed knowledge base such that no agent has complete knowledge of the network organization and structure.

An important feature of the system is cooperation of agents. Two general methods for cooperation have been explored. First, problem solvers may cooperate through the exchange of messages. Agents may coordinate their actions by requesting another agent to perform some task in order to achieve a goal. Cooperation of agents may also take the form of an exchange of knowledge through messages when one agent needs additional knowledge in order to further problem solving activity. The second mechanism for cooperation is through sharing local knowledge about the current state of the network and the status of problem solving activity. Inferences of one agent are shared with the others in a central knowledge base. The shared knowledge base is managed by a knowledge base manager.

In order to implement this architecture in a laboratory environment, we developed a Distributed AI System (DAISY) testbed which supports simulation of multiple agents on one or more LISP processors. The DAISY testbed incorporates two system building tools which we developed during this effort. SIMULACT is a generic tool for simulating multiple actors in a distributed AI system. It is domain independent, permits rapid prototyping of distributed software modules, and allows interactive experimentation incorporating a gauge facility for monitoring and data collection. A

graphical user interface (GUS) was developed to assist a user in capturing the structural knowledge about a communications system. A typical communications network consists of thousands of objects which are interrelated by the structural organization of the network. GUS enables a communications network expert to describe the details of this organization using a familiar set of graphical symbols within an easy-to-use interactive window environment.

Our primary results have been in the development of cooperation paradigms for distributed agents. As an example of cooperation through message exchanges, the service restoral task requires agents which perform distributed planning subject to constraints imposed by network topology and resource availability. We have developed a distributed multi-agent planner (DMAP) which extends the current work in planning by designating certain objects as resources so that they may be efficiently allocated for effective use in satisfaction of multiple goals. The planner consists of two stages, plan generation and multistage negotiation. During plan generation, agents are required to generate plans which utilize limited system resources in a domain where both the knowledge about resources and the control over these resources are distributed among the agents. Experimental results are presented which show that plan generation in this class of problems can be accomplished by the exchange of a limited amount of information between agents. It is unnecessary for any single agent to acquire complete global information about the system.

After a set of plans has been established, agents must cooperatively select specific plans to execute as many goals as possible, subject to resource constraints. Multistage negotiation has been developed as a means by which an agent can acquire enough additional knowledge to reason about the impact of local decisions on nonlocal system state and thus modify its behavior accordingly. Because no single agent is in control and no single agent has complete knowledge of the entire system state, an important aspect of multistage negotiation is the mechanism for providing agents with nonlocal information. We have developed a formalism for abstracting and propagating information about the nonlocal impact of decisions made locally. Our work provides mechanisms for determining impact at three levels: locally on the level of plan fragments, locally on the level of goals, and nonlocally. This approach may be viewed as promoting cooperation among agents by using constraint-based reasoning to develop good, local heuristic decision making.

A second example of our results in the study of cooperation methods involves sharing knowledge among local agents. We have implemented a method by which knowledge can be shared in a local knowledge base in the form of inferences and default assumptions. Specifically, a Multiagent Assumption-based Truth Maintenance System (MATMS) has been developed to manage a knowledge base shared by multiple problem solvers. The most important feature of the MATMS is that it provides the foundation for resolving inconsistency between agents, while supporting the notion

that two problem solvers can have different views concerning the state of a particular piece of knowledge. The MATMS handles differing views by allowing independent belief sets for each of the agents. It supports resolving inconsistency between agents by providing a mechanism for comparing two agents' belief sets. An agent's belief set is characterized as the default knowledge base (which is common to all agents) with an overlay placed upon it. The MATMS is efficient largely because it focuses its efforts on managing these overlays, not the entire belief set of an agent. By concerning itself only with the overlays, the MATMS can switch from addressing one problem solver's belief set to addressing another's expeditiously. It can also change an individual problem solver's belief set quickly, because the default knowledge is not explicitly carried over from one belief set to another.

The third area of significant results for distributed cooperation is the development of a distributed automated reasoning system (DARES). This represents some of our most recent work and thus is still in an early stage of development. DARES has implemented a distributed theorem prover as a model for cooperation among agents performing a distributed situation assessment task. Each agent has some limited view of the network state and thus may form an hypothesis of the global network state. However, in order to confirm this view, additional knowledge obtained from other agents is necessary. Confirmation of this view becomes a theorem to be proved. Our preliminary results indicate this approach appears to be very promising.

In a series of tests which investigated comparisons of distributed reasoning tasks using DARES, we found that a distributed set of agents with *only partial knowledge* performed better than a single agent with complete knowledge, or a similar set of agents each having complete knowledge. The reason for this behavior is that having complete knowledge often expands the search space without providing a compensating means for focusing the search. In a multi-agent system with each agent having only a limited view, each agent is able to focus its own search more quickly. Messages are only exchanged after an agent has made some progress and thus narrowed the space of relevant responses. We believe these test results provide solid evidence in support of our architectural approach which is to employ distributed agents reasoning from a localized and limited perspective and cooperating in finding global solutions.

The building of an infrastructure to support the continued growth of AI research activities among the NAIC universities was an important ancillary objective of this research project. At Clarkson we have made significant developments in terms of the growth of faculty activity in AI research, offering of new AI courses at both undergraduate and graduate levels, addition of new research facilities, and an increase in the numbers of graduate students working in AI. During the past five years we have formed new working relationships with research groups at other NAIC universities and with other contractors to RADC. These relationships have strengthened our abilities to conduct meaningful research and to assist the transfer of technology from the

university research lab to the industrial development environment.

Finally, this report looks ahead to future research problems. While we believe we have made significant progress during the past five years, the work started then is not yet complete. As a result of what we have learned about knowledge representation and acquisition, we have new ideas for the design of a local knowledge base for network management. The importance of graphical tools was underestimated in our original design of the DAISY testbed. Since the human operator or user is expected to continue to play an important role in these systems, an improved interface between intelligent agents and the human is critical to the development of a successful system. With the basic testbed now defined, and component parts developed, there remains much work to be done in testing the initial design choices under a variety of adverse operating environments. We have yet to determine the robustness of our cooperation paradigms under stress. Thus this report should be looked upon not as the conclusion of this work, but rather as having set the stage for new work in investigating the design of distributed AI systems. These systems have application not only in communications network management, but also in other areas of command, control, communication, and intelligence information processing.

4.2 Introduction

This report documents the primary results of a five-year research project which investigated the application of distributed artificial intelligence (DAI) to the management and control of communication networks. This work has produced three broad categories of results: (1) development of a model architecture for a distributed, intelligent network management system, (2) design and implementation of a distributed AI system testbed, and (3) implementations of three distinct DAI systems. The important features of this problem domain and the scope of research issues investigated are introduced in the remainder of this section.

4.2.1 Problem Domain Description

The application domain of interest for this research effort is the monitoring and control of large communications systems. Maintaining reliable communications under a wide variety of operating environments is vital to the preservation of both our national security and world peace. It is an important problem to the Department of Defense, and as we will discuss in the paragraphs which follow, it provides a rich set of problems for research in distributed problem solving.

In our studies we have concentrated on the Defense Communications System (DCS), and especially on the European theater. The DCS is a highly complex system consisting of tens of thousands of circuits interconnecting users at more than

300 sites world-wide. We have chosen the European theater for several reasons. The DCS network structure in Europe is particularly interesting for the study of distributed problem solving paradigms. It consists of a large number of sites (about 200) which are interconnected in an irregular structure. It is currently controlled by close cooperation and coordination among a group of highly skilled human controllers distributed throughout the system. The variety of transmission media and communications equipment in use give rise to the need for sophisticated problem solving tools to assist these human operators in providing the best possible control of the system.

The size, complexity, and near constant state of change of the DCS make it unwieldly for direct incorporation into our investigations. Instead, the goals of this research program have been better served by using a simplified model of the DCS which incorporates those characteristics important to system control. This section describes the organization of the DCS and the tasks involved in system control. Emphasis is placed on those system features and aspects of problem solving activity relevant to our model of the DCS.

4.2.1.1 Organization of the DCS The DCS is a large, complex communications system consisting of many component subsystems. It provides the long-haul, point-to-point, and switched network communications needed by the DoD. A careful analysis of the DCS reveals that the organization of the DCS must be viewed from a multidimensional perspective. For example, all DCS facilities may be divided into one of two groups: either DoD-owned or DoD-leased. As a general rule, the majority of DCS facilities in the continental U.S. are leased, whereas the majority of facilities overseas are owned and operated by the DoD.

The DCS may also be viewed as a layered organization consisting of three basic layers: transmission facilities, circuits and networks. Each of these layers may be further subdivided into component subsystems. Transmission facilities may be either terrestrial or satellite. Terrestrial transmission is based on either analog or digital channels, multiplexed into groups or digroups, and then into supergroups which are transmitted over communications links from one station to another. The most common transmission medium used is line-of-sight (LOS) microwave; however, there are also tropo-scatter, fiber optic, and cable links used. Satellite transmission facilities are also used, primarily for transoceanic links. We have not used satellite links in our model.

The transmission facilities form the backbone structure over which the second layer, consisting of circuits and trunks, is built. The European theater of the DCS consists predominately of dedicated circuits between users. These circuits may traverse several stations following fixed paths. There are a number of key data items which are associated with each individual circuit or trunk, and which are important in the performance of system control. These are important details in our model,

and include the user priority level, the restoration priority, and the quality of service required.

Networks form a third layer of the DCS organization. There are three general categories of networks: voice switched, data switched, and dedicated or special purpose networks. These networks rely on trunks to provide the interswitch connectivity. The voice networks are AUTOVON, AUTOSEVOCOM (a secure voice network), and DSN (Defense Switched Network). These networks provide circuit switched voice connections among subscribers. The data networks include DDN and AUTODIN. These networks are in a period of evolution from the older AUTODIN style network to the modern, packet switched DDN style network. At this stage of development we have not incorporated details of networks into our model.

Yet another perspective of the DCS is equipment oriented. The DCS consists of a very large inventory of communications equipment, such as modems, multiplexers, radios, switches, etc. Each equipment item has certain distinguishing characteristics including its function within the overall system, its status signals (which may be monitored and made available to system controllers), and its control capabilities (which provide the mechanism for implementing desired control actions on the system). Knowledge about equipment is vital to problem solving agents attempting to control the system, and cuts across the layered organization described above. For example, a particular multiplexer may be a part of a transmission facility, as well as a part of one or more circuits, and a part of one or more networks. Generic equipment types incorporating the important characteristics of typical units in actual use have been used in our model.

The final dimension along which the DCS may be analyzed is its organization for monitoring and control. Currently the DCS system control function is almost entirely manual, and is highly fragmented. Each new network, or transmission subsystem incorporated into the DCS has included its own control system. As the DCS evolves to a modern, digital communications system, with automated control systems, it has become increasingly important to integrate these various controls. In the next section we discuss the system control problem. Our view of DCS system control is based on our understanding of the future directions system control for the DCS will take.

4.2.1.2 System Control of the DCS System control is defined [17, page 2-1] as the process "... which ensures user to user service is maintained under changing traffic conditions, user requirements, natural or manmade stresses, disturbances, and equipment disruptions on a near term basis." System control incorporates five major functions: facility surveillance, traffic surveillance, network control, traffic control, and technical control. Each of these functions will be described in more detail and related to specific problem solving activities in the paragraphs which follow.

DCS system control is to be organized in a five level hierarchical structure. Be-

ginning at the lowest level and moving up, each level in this hierarchy represents a broader view of the DCS, a larger geographic area, a greater responsibility and a higher authority. Level 5 (the lowest level) represents stations or facilities at which either a technical control or patch and test capability exists, or an access switch exists, or an earth terminal for a satellite link exists. Level 4 represents either a major technical control facility or nodal switch. Level 3 represents a subregion control center (SCRF). Level 2 corresponds to theater level control and may be either an area communications operations center (ACOC) or alternate ACOC. Level 1 is the worldwide Defense Communications Agency Operations Center (DCAOC). For the purposes of our research, we are concerned with level 3 and lower levels. These are the levels most closely associated with the real time or near real time control of the system. Within the European theater approximately six SRCFs are expected to be established. Each of these control centers will have responsibility for integrated control of transmission, circuit, trunk and network resources over a significant portion of the DCS. Thus, it is at this level (level 3), or lower, that the need for cooperative problem solving is likely to be the greatest.

Three distinct problem solving activities have been identified within the five major functions of system control. We refer to these activities as *performance assessment* (PA), *fault isolation* (FI), and *service restoral* (SR). A general task description for each of these is given below and related back to one or more of the five functions of system control.

Performance Assessment (PA)

Performance assessment may be viewed as a problem in data interpretation and situation assessment. Since data is available only on a distributed basis, coordination must take place among the PA agents in order to arrive at a coherent view of the state of the communications system. The facility surveillance and traffic surveillance functions of system control are included within the PA activity. Real time equipment, transmission network, and traffic data are measured and collected to provide the controller with the information needed to determine the status of the transmission system and facilities, the quality of communications circuits and network performance. Trouble reports from users are also significant inputs to this activity.

The goal of PA is to formulate a local view of system status and performance, and to identify as quickly as possible the impact of any observed deviations from normal operating conditions. The PA agent is responsible for determining the need to invoke either fault isolation and/or service restoral agents. Since few problems are likely to be localized within the area of responsibility of a single SRCF, the PA agent must also communicate with similar agents in neighboring areas to arrive at a consistent assessment of status throughout the system.

Fault Isolation (FI)

The fault isolation task is a diagnostic activity. It is concerned with identifying the specific cause and location of faults within the communications system. The term *fault* is used in a very broad sense to mean either a complete outage of service or a degradation in quality or performance. The FI agent responds to reports of known or suspected faults determined by PA. Much of the same data available to PA is also used in the FI activity, however the analysis is carried out in greater depth, and the cause-effect relationship is emphasized. In some instances the immediate results of FI may be inconclusive and will require additional testing to resolve ambiguities in the data.

The FI agent incorporates the in-depth analysis aspects of facility and traffic surveillance as well as the testing aspects of technical control. Coordination and cooperation with similar agents at intermediate or distant end stations involved in a faulty link, trunk, circuit, or network are often necessary to determine the cause and location of a fault.

Service Restoral (SR)

Service restoral is a plan generation activity which recommends a set of specific control actions needed to restore user service. These actions may involve alternate routing of trunks or circuits, switch control, or transmission system configuration control (such as reallocation of equipment, use of backup or spare equipments, etc.). The network control, traffic control, and technical control functions are encompassed in the SR activity.

4.2.2 Scope of Research

Clearly a complete and integrated approach to an intelligent, distributed network management system would be a major project involving basic and applied research, prototype development, field testing, and eventual full scale production. Such an effort is, in fact, not feasible as a single project given the dynamic advancement of computer-communications technology. Rather, as the DCS technology evolves, we should expect system control to advance along with it. It seems clear that this advancement will likely include some sort of knowledge-based, expert or intelligent systems. The objective of this research was to investigate the issues which arise in the application of distributed intelligent systems to communication network management.

The scope of this work involved the development of a simplified model of the DCS network, an analysis of functional requirements for problem solving, a study of knowledge representation for these problems, and development of an architecture

which incorporated the necessary problem solving skills. In order to construct and test various distributed components of this architecture, we designed a general purpose distributed AI testbed which greatly facilitates program development and simulation of multiple intelligent agents in a distributed system. As mentioned previously, our results may be viewed in terms of their contributions to the state of knowledge in distributed artificial intelligence and to the development of future distributed network management systems. These contributions are described in detail in the remaining sections of this report.

4.3 A Model for Distributed, Intelligent Network Management

An important contribution of this research to the future development of distributed network management systems has been the creation of a model for a distributed, intelligent network management system. This model is based on an architecture involving a functional distribution of agents at a single site, together with a spatial distribution of similar agents across a network of control centers. The development of this architecture also includes the design of a distributed knowledge base.

4.3.1 System Architecture

In this section, we present the design of an architecture for an intelligent distributed problem solving system to assist in the management and control of a communication system such as the DCS. This discussion addresses three primary facets of the architecture: (1) the role of intelligent information processing in the context of the overall system, (2) the structure of the intelligent system residing at each node, and (3) the overall structure of the distributed problem solving system.

Any deployed distributed problem solving system would most likely be viewed as an intelligent assistant to the system controllers in the field. It would perform the tasks of filtering the data received, analyzing and interpreting it as well. Based on these interpretations, it would recognize various critical situations which might arise in network operations; it would advise human operators as to which equipments are probably malfunctioning; and it would suggest alternative restoral plans and appropriate control actions. The human operator would have the responsibility of directing service restoral, dispatching service personnel, and initiating control actions. In addition, the human retains the privilege of preempting the system at any time. As an intelligent assistant, the system would relieve the controller of many tedious tasks. It would also provide a vehicle for training activity that can be utilized by new personnel.

Distributed problem solving systems are often viewed as being comprised of a

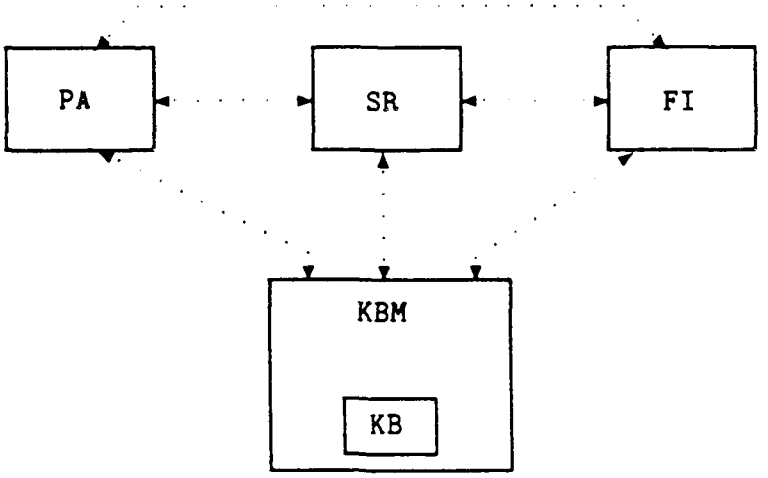


Figure 1: Control Node Architecture

network of loosely coupled agents which cooperate in solving a problem. In the context of communication systems, each locus of control activity is also a site where a node of the problem solving network resides. As the discussion in the previous paragraph indicated, several problem solving activities may be active concurrently. For example, performance assessment would typically be an ongoing data filtering and interpretation task, while fault isolation and service restoral tasks would be initiated in response to specific events. We have chosen a node level architecture that represents a decomposition of nodal problem solving activity into these three primary tasks.

The specific structure of the node level architecture is shown in Figure 1. There are three primary problem solving modules at the node level: Fault Isolation (FI), Service Restoral (SR) and Performance Assessment (PA). Each of these modules or agents requires access to the same knowledge about the structure and expected behavior of the network being controlled; this knowledge is contained in the local Knowledge Base (KB). There are two mechanisms for cooperation among these local agents: exchanging of messages and sharing of inferences. Messages may be exchanged as shown in the diagram by interagent communication paths (dotted lines in the figure) in order to coordinate specific actions, such as the request by one agent for another to complete some task. Inferences are shared by sharing a common local knowledge base. In order to coordinate multiple agents' access to this knowledge base, a Knowledge Base Manager (KBM) is needed. Our work in DAI has included the development of a domain independent multiagent truth maintenance system, MATMS, which forms a significant component of the KBM. The MATMS will be described further in Section 4.5.2.

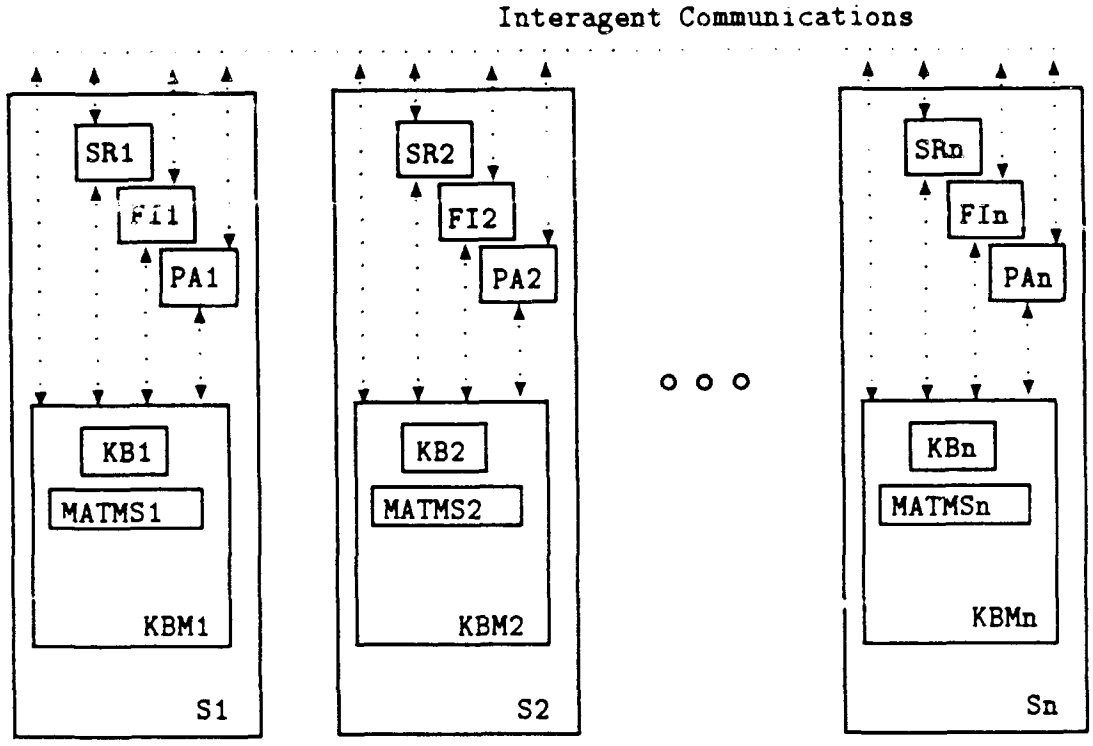


Figure 2: Distributed System Architecture

The overall structure of the distributed problem solving system reflects distribution in two dimensions. At one level, the system is seen as a number of complex agents that operate in a loosely coupled fashion to solve the problem of controlling the communication network. At another level, the overall system can be regarded as a group of relatively independent, specialized, distributed problem solving subsystems cooperating in the solution of one or more similar type problems. One of these subsystems is composed of the group of fault isolation agents. The fault isolation agent at each node cooperates with its counterparts at other nodes in solving the fault isolation problem for the communication system. In a similar fashion, the service restoral and performance assessment agents can be regarded as distributed problem solving subsystems in their own right. This architecture is shown in Figure 2. Each large block, denoted by Sn, represents the local intelligent problem solving system at a single control site. The dotted lines indicate interagent communications. We observe that at this level, all cooperation must take place by message exchange since geographic separation makes sharing a common knowledge base impractical.

Our results in designing, implementing, and testing a distributed system archi-

tecture will be presented in subsequent sections of this report in the form of two distributed reasoning systems, DMAP and DARES. Our results using DMAP (cf. Section 4.5.1) has shown that plans for service restoral can be generated in a distributed fashion with no central control and no global knowledge of network configuration. Further, DMAP has demonstrated that distributed decision making can lead to a globally coherent and globally optimal selection of restoral actions by the exchange of messages which describe the nonlocal impact of local decisions. Although DARES (cf. Section 4.5.3) is at an earlier stage of development than DMAP, we believe it demonstrates a means for cooperation among a wide variety of "expert system-like" agents by the exchange of knowledge in the form of inferences obtained as partial results in a theorem proving context. This cooperation paradigm is being used in implementing distributed performance assessment agents.

4.3.2 Knowledge Representation

In forming the model discussed in the previous section, we discovered there were two primary categories of knowledge: problem solving knowledge, which was largely domain independent, and domain knowledge. The domain knowledge for a physical system such as a large communication network is primarily structural knowledge. Structural knowledge of a physical system embodies the components of the system, the behavioral characteristics of these components, component connectivity, and system behavioral characteristics derived from component behavior propagated along connections [4]. Example domains of which structural knowledge is an inherent property include communication networks, automated factory configurations, and electrical circuits. Knowledge about the structure of a physical system is needed by a wide variety of problem solving tasks. Our analysis of problem solving activities, such as fault isolation, service restoral, and performance assessment, found that each depend upon structural knowledge to reason about the network.

Natural extensions from structure are behavior and function. Informally, system structure refers to how system components are connected, behavior refers to how these components act, and function refers to the purpose of these components. The behavior of a system is expressed in system terms, whereas the function of a system is expressed in terms relating the system to its environment. For example, the typical behavior of a traffic light is to display a red, yellow or green light. The function of a traffic light is to control the flow of traffic.

The design of a knowledge representation scheme should not only concern structural details, but also address behavioral characteristics. This is particularly important since the principle problem solving activities in this domain (fault isolation, service restoral and performance assessment) rely heavily upon component and system behavior as well as system structure. Our knowledge base must embody both structural and behavioral knowledge.

Ongoing research relates a physical system's structure to its function by traversing the intermediate system property of behavior [29]. It is this transition from structure to function via behavior which is going to relate the machine captured structural knowledge to machine control of system behavior.

We have designed a knowledge representation scheme which supports the problem solving activities performed in a distributed network management system. The knowledge base at a local control site contains information about the structure, function and current state of the network, and will be shared by the various problem solving agents at that control site.

An important feature of this local knowledge base is that it forms a component in a network-wide distributed knowledge base. In distributing other aspects of network management, such as data collection, data reduction, decision making, and control action execution, we believe the knowledge base must be distributed as well. Since the knowledge base includes a great number of details about local equipment configuration, allocation, and status, it seems highly unlikely that this level of detailed information should be widely known outside the primary area where it is generated. The local knowledge base must, however, include less detailed, partial views of the non-local parts of the network.

4.2.2.1 Domain Knowledge Classification The domain knowledge base contains three types of knowledge: graphical knowledge, structural knowledge and state knowledge (Refer to Figure 3). Graphical knowledge is the primary mechanism for the graphical representation of structural knowledge; in the current system, graphical knowledge is only used during user input or editing of the knowledge base. Structural knowledge embodies configuration knowledge and communication path knowledge, each of which entails the representation of application domain objects and how they are physically related. State knowledge represents self-descriptive attributes and status of application domain objects. The key point to remember here is that knowledge about structure and state is common and available to each of the different problem solving activities at a local control site.

As shown in Figure 3, there are three levels of configuration knowledge corresponding to the natural hierarchy of application domain structure: subregion, network, and equipment configuration knowledge. Configuration knowledge is largely topological in nature, and describes the physical communications system. Knowledge about this system is hierarchical in nature, where at the highest level subregions are represented in terms of one station designated as the current SRCF or SubRegion Control Facility. This is the control site mentioned previously, and it is responsible for the control decisions made for the stations within its subregion, or area of responsibility. At the network level stations are interconnected via links, and collections of stations are grouped into subregions. Each station corresponds to a node or site in our distributed

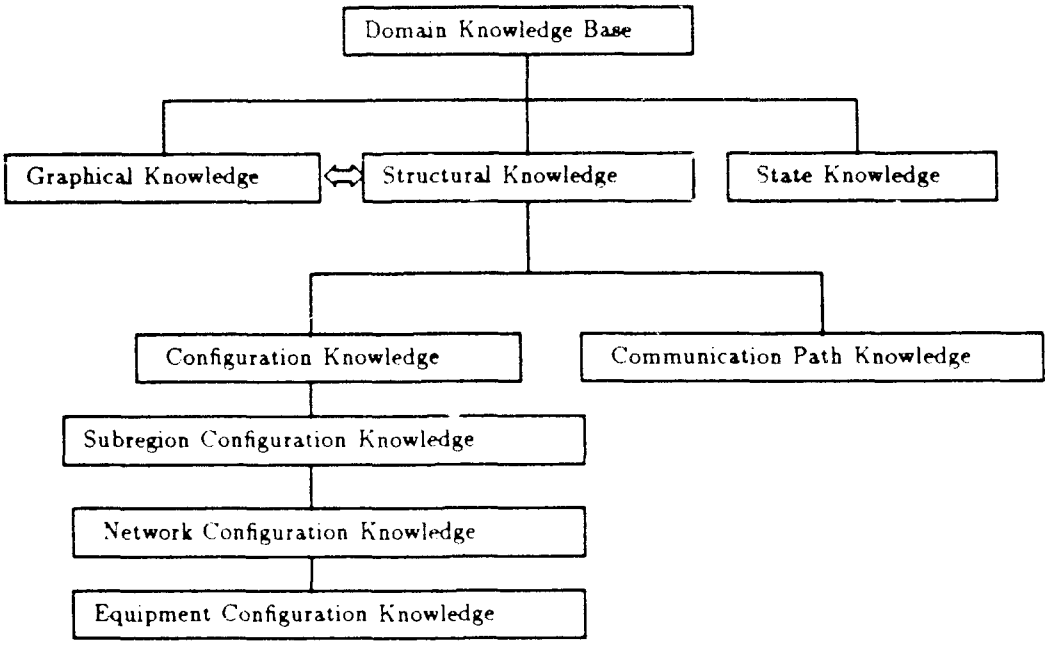


Figure 3: Classification of Knowledge

problem solving system. The knowledge base therefore contains complete declarative knowledge concerning each site in the subregion, including its name, geographic location, and its current operating status. In addition, each link in the subregion is also represented, with information concerning which sites correspond to the link's two endpoints, and the media type and capacity of the link.

At the third level in the hierarchy, within each site there is a collection of interconnected equipments. The physical topology is similar to the top level network, but at this level specific pieces of equipment are interconnected by various kinds of arcs. A typical equipment configuration for a site is given in Figure 4. Types of equipment include radios, second level multiplexors (MUX), first level MUXes, and digital patch and access systems (DPAS). Interconnecting these equipments, and connecting equipments with users, are supergroups, digroups, and channels. The equipment configuration knowledge within each site also forms a hierarchy. A radio within a site may be connected to a second level MUX or to another radio at the same site via a supergroup. Each second level MUX is connected either to a first level MUX or to a DPAS via a digroup. Finally, each first level MUX is connected to a user via a channel.

In addition to the knowledge described above concerning specific pieces of equipment and their connectivity, equipment configuration knowledge also contains more

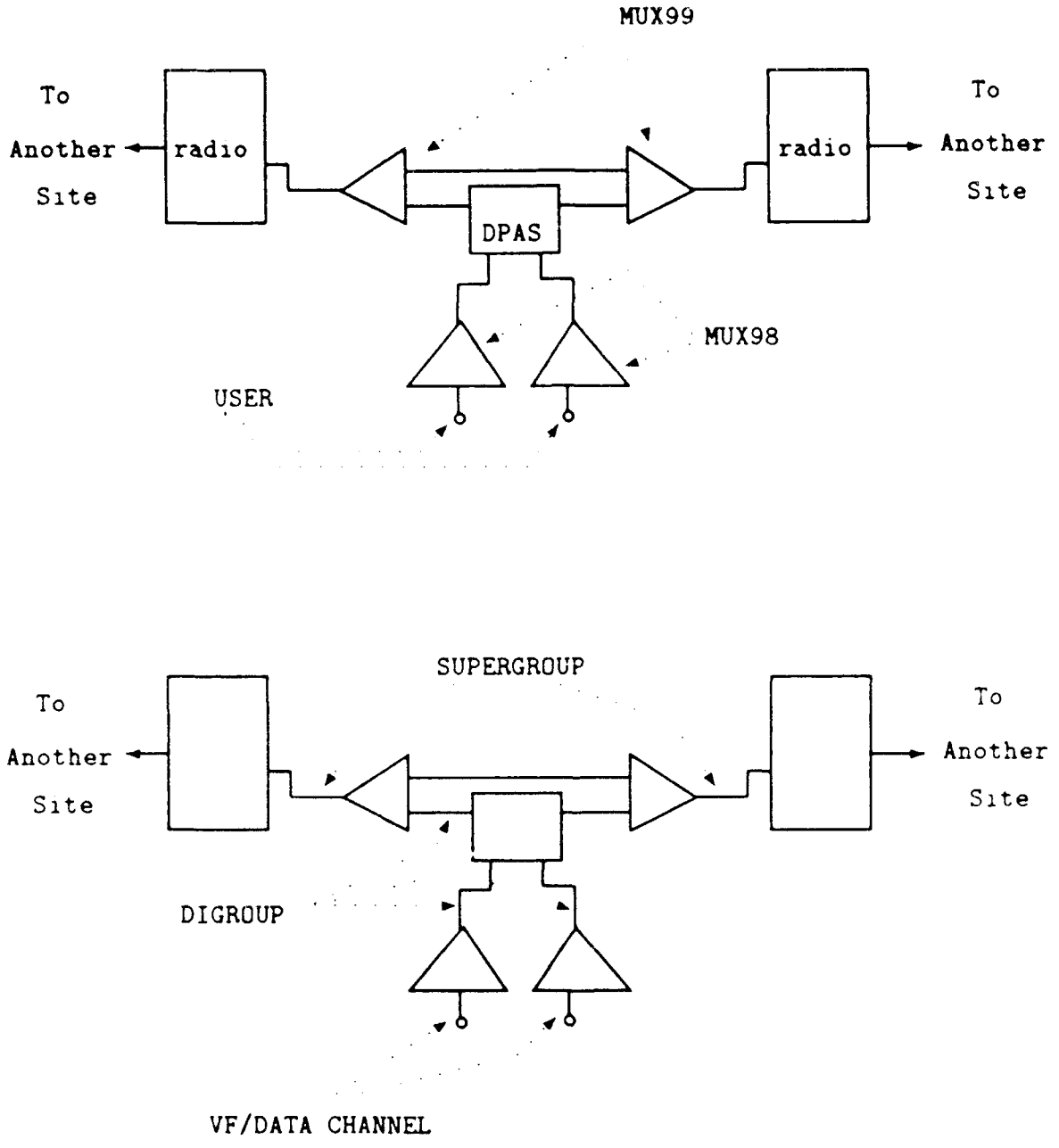


Figure 4: Typical Equipment Configuration

general information on each item in the hierarchy: that is, generic information on subregions, stations, links, as well as kinds of equipments and connections among them is represented in the knowledge base. For instance, generic information common to all radios of the same type includes the type of alarm signals which may occur, specific actions to take in the advent of such alarms, and operational parameters for that kind of radio.

The configuration knowledge describes the components and connections which comprise a topology representing the communication network system as a whole. This network structure forms a natural guideline for search of solutions in many instances, for example, in service restoral. This structural hierarchy permits abstraction of the search space which will expedite search techniques.

Equipment state knowledge is also necessary for system control. Whereas equipment configuration knowledge describes the physical connectivity of the network, equipment state knowledge contains information concerning the current operational status of each piece of equipment. This includes any current alarm signals for the equipment, but also includes expected behavioral characteristics for each kind of equipment, and any deviations from this norm which a specific piece of equipment might have. Equipment state knowledge also incorporates conclusions and interpretations of the measured status data as produced by an agent such as performance assessment or fault isolation. This data is available to any agent as it analyzes the current state of the communications network.

Finally, communication path knowledge reflects the current user-to-user connections in the network. The primary unit of telecommunications service carrying message traffic between two locations is known as a communications channel. The three telecommunications connectivity entities are circuits, trunks and links, which carry the transmitted signals in a communications channel. A circuit is a path between two end-users, and consists of a sequence of adjacent nodes and arcs such that no node or arc appears more than once in the path. A node corresponds to a location where a communications signal may be originated, manipulated, or terminated, and an arc consists of the set of communications channels between two adjacent nodes. Examples of nodes include end-user points at some location, drop-and-insert points which correspond to intermediate nodes in a communications channel, and PTT-pickup points where a communications channel is transferred from a DCS controlled transmission facility to a common-carrier transmission facility. A trunk is a single communications channel between two or more nodes, and may itself be channelized: a trunk or any of its channels may carry a single circuit or another trunk, and the signals at the initial node and the terminating node of the trunk are in the same form. Finally, a link is as described above in the equipment configuration knowledge. It is a transmission facility, such as a cable or microwave radio system, connecting two adjacent nodes, and may be channelized. The terms node, arc, and path are related to geographic

terms location, link and route; that is, a route is a sequence of transmission facilities traversed by a communications channel, and so can be described by a sequence of locations and links. Also, communications path knowledge is time-dependent, as circuits and trunks are associated with a particular configuration only for a finite period of time, and may be alternately routed if necessary.

4.3.2.2 Design of the Knowledge Base The design of the local knowledge base is built on two fundamental ideas for knowledge representation: frames and inclusion hierarchies. It was implemented using an object-oriented programming paradigm.

A frame-based system was used because it provides a natural way to describe the many details about each equipment type, link type, or site. A frame representation groups each of the significant characteristics of an object type together while providing a modular structure for the knowledge base. This modularity is of great importance in developing efficient search methods for very large knowledge bases. In our work we have taken advantage of this modular organization in the development of local agents for performance assessment and service restoral. The frame structure also facilitates the incorporation of default values for equipment configurations and operating status as a substantial portion of the knowledge base. In designing the MATMS truth maintenance system, which forms part of a knowledge base manager, we found the frame representation particularly appropriate.

The second significant characteristic of the knowledge base design is the use of inclusion hierarchies in representing network knowledge. The structure of the knowledge suggests an inclusion hierarchy for two reasons. First, there are numerous examples of knowledge about generic equipment, such as radios. Given specific examples of these objects, it is useful to transfer the known properties and attributes of the generic object to the specific example. This is known as inheritance. The second reason for using inclusion hierarchies is the natural hierarchy of communication paths imposed by the multiplexing schemes in common use. Thus we have inheritance of properties among physical objects (radios, multiplexers, etc.) and also among logical objects (links, trunks, circuits, etc.).

This knowledge base is implemented in ZetaLISP on a Symbolics 3670 Lisp machine. Object-oriented programming techniques were employed to describe the abstract data types needed, and to provide data encapsulation. This provided the modularity and flexibility needed in a developing system. In addition, since many of the objects have homologous features, the implementation makes extensive use of the concept of methods or generic functions. Methods are used to implement common functions to be performed on similar objects. For example, there are common operations to be performed on equipments, such as radios, multiplexers and switches, or on path components, such as links, supergroups, and digroups.

Further implementation details are provided in Section 4.4.2 which describes a

graphical tool for user interaction with the knowledge base.

4.4 Development of a Distributed AI Testbed (DAISY)

We have developed a Distributed AI SYstem (DAISY) testbed which supports simulation of multiple agents on a group of heterogeneous LISP processors. The DAISY testbed incorporates two system building tools which we developed during this effort. SIMULACT is a generic tool for simulating multiple actors in a distributed AI system and is described in Section 4.4.1. In Section 4.4.2 we describe a graphical user interface (GUS) which assists a user in capturing structural knowledge about a communications system.

4.4.1 Distributed Environment Simulator (SIMULACT)

As part of the testbed, we have developed an environment intended to aid the development of applications involving distributed problem solving. Specifically, we will describe in this section a domain independent development and simulation facility which permits rapid prototyping, interactive experimentation, and ease of modification of such systems.

Our environment, called SIMULACT, is based on a model which regards intelligent agents as semi-autonomous problem solving agents which interact with one another by means of a message passing paradigm. This system is currently implemented on a network of LISP machines which incorporates both TI EXPLORER and SYMBOLICS machines.

Distributed problem solving systems have received increasing attention in the AI community. Two factors have motivated this phenomenon. First, the advent of large parallel machines and the development of small, powerful microprocessor based systems have encouraged research on problems related to parallel and distributed AI systems. Secondly, research in distributed problem solving has been driven by the observation that a number of important applications are inherently distributed. Examples include distributed situation assessment, distributed sensor nets, air traffic control, and control of geographically distributed systems such as communications systems and power transmission networks.

It is easy to envision application environments in which on the order of ten to fifty semi-autonomous agents might be cooperatively solving a problem. In such an application, a distributed problem solving system would generally be implemented as a distributed system with as many independent processors as there are agents in the system. It would be prohibitively expensive to build a network of processors for the purpose of providing a testbed in which feasibility studies could be performed and initial prototype systems developed. The alternative of simulating the desired system

on a single processor is not attractive, since testing a system of this magnitude on a single LISP machine would probably require time consuming simulation runs for evaluation purposes. A facility which permits a network of k machines to emulate the behavior of a network of n machines (where $n > k$) would provide an attractive alternative.

SIMULACT is a development environment for distributed problem solving systems which provides such a facility. The underlying model of problem solving which is employed regards the problem solving system as a collection of semi-autonomous agents which cooperate in problem solving through an exchange of messages. The system is modular: each agent is essentially an independent module which can easily be "plugged in" to the system. An agent's interaction with other agents in the system is totally flexible, and is user specified. Neither the form nor the content of inter-agent messages is specified by SIMULACT itself. In addition, the user can suspend execution at any time, examine the state of any agent, modify the state, the knowledge base, or even the code of an agent, and resume execution. A trace facility makes post-mortem examination of activity feasible, and a gauge facility allows the user to instrument the system in a very flexible manner.

4.4.1.1 Background The architectures of the distributed AI systems that have been developed have largely been driven by the nature of the application. For example, the DVMT [33] is clearly a descendent of the HEARSAY systems, and this has been natural because the signal interpretation tasks involved in vehicular tracking are similar in nature to those of speech recognition. There have been very few efforts directed towards the problem of establishing a domain independent environment suitable for the development, testing, and debugging of distributed applications.

One of the notable exceptions is MACE (Multi-Agent Computing Environment) [21]. This system is a development and execution environment for distributed agents. It essentially provides a tool for programming multiprocessor systems in an object oriented fashion. In MACE, agents are regarded as intelligent entities, each of which is capable of performing tasks. These agents are directed and organized by the programmer through the specification of inter-agent relationships together with high level directives and constraints on behavior. Agents in MACE are implemented as property lists of the agent name, so only one copy of an agent may reside on a given processor. By contrast, in our system multiple instances of a given agent type can be resident on a processor, thus providing greater flexibility to the user.

Among those systems that are intended for development of distributed applications, most have been designed with the intent of gaining as much speed in execution as possible. Examples of this type of system are found in Stanford's CAGE and POLIGON. Both of these systems are designed for parallel execution of applications built using them and are based on a blackboard model of problem solving. CAGE

[1] provides a problem solving framework which is based on the assumption that development will proceed on a multiprocessor system involving up to a hundred or so processors with shared memory. The user must specify explicitly what is permitted to run in parallel. POLIGON [40], on the other hand, is designed to be run on distributed memory multiprocessor machines involving a large number (hundreds or thousands) of processors. High bandwidth interprocessor communication is necessary for a successful implementation of POLIGON. A number of primitive operations (such as rule evaluations and blackboard updates) are automatically done in parallel.

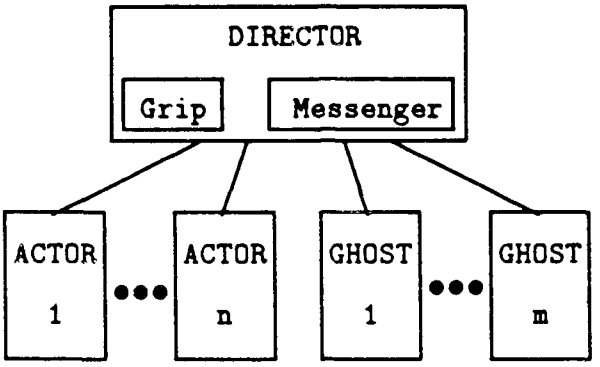
Both CAGE and POLIGON have programming languages associated with them. As in our system, these languages facilitate the writing of application programs and provide a layer of abstraction between the user and the system's implementation details. Unlike our system, CAGE and POLIGON have been devised in an attempt to investigate issues related to exactly how much speedup can realistically be anticipated when AI programs are run in parallel environments.

In the following two sections, we discuss SIMULACT's system structure and concurrency control mechanisms. We also describe the five user interface facilities, which were designed to make SIMULACT attractive as a development environment. We conclude with a brief discussion of experiments we have performed in order to assess the degree of overhead due to SIMULACT as implemented on one and two machines.

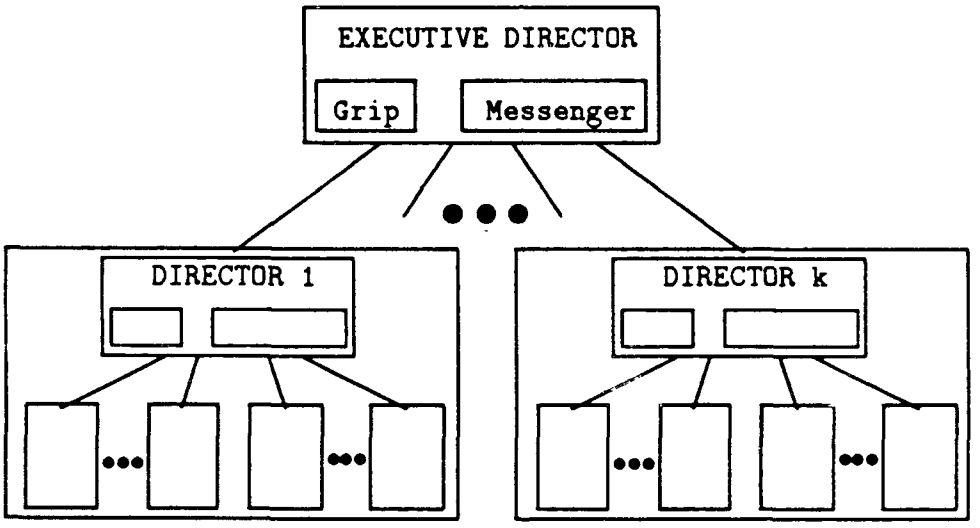
4.4.1.2 System Structure SIMULACT is a distributed system that allows n agents to be modeled on k machines, where $n > k$. Each agent runs asynchronously and coordinates its activity with that of other agents through the exchange of messages. The activities performed by each agent are assumed to be complex, so that the parallelism is coarse grained. SIMULACT allows the programmer to write code in Lisp as though there were as many Lisp machines in the network as there are agents in the distributed system being developed.

As is evident from Figure 5, SIMULACT is comprised of four component types: Actors, Ghosts, Directors, and an Executive Director. Actors are used to model agents in the distributed environment. Each Actor type is individually defined, and used as a template to create multiple instances of that Actor type. An Actor is a self contained process which runs in its own non-shared local environment. Although Actors run asynchronously, the elapsed CPU time for each actor never varies by more than one *time frame*. These Actors closely resemble the entities described in Hewitt's "Actor Approach To Concurrency" [23].

Ghosts are used in SIMULACT to generate and inject information into the model that would naturally occur in a "real" distributed expert system. They do not represent any physical component of the model. For example, external inputs (alarms, sensors, etc.) affecting the state of the system can be introduced via Ghosts, as well as inputs that reflect the *side effects* of the systems activities. Ghosts can also be



a) Host level structure



b) Network level structure

Figure 5: SIMULACT's modular structure

used to inject noise or erroneous information into the system so that issues concerning robustness can be easily investigated. The performance of an expert system can be monitored in subsequent runs through the simple modification of these Ghosts.

Due to the similarities between Actors and Ghosts, we refer to them as Cast members. Each Cast member has a unique **stagename** and a **mailbox** used by the the communication facility in routing messages among members. Each also has a **script function** which defines its high level activity.

The control structure residing at each host processor in SIMULACT's distributed environment is known as the Director. The Director is responsible for controlling the activities of the Cast members at that site, and for routing messages to and from these members. These activities are assigned to the Grip and Messenger respectively. The responsibilities of the Grip range from setting up and initializing each Cast member's local environment to managing and executing the Actor and Ghost queues. The Messenger only deals with the delivery and routing of messages. When a message is sent, it is placed directly into the Messenger's *message-center*. During each time frame, the Grip invokes the Messenger to distribute the messages. Whenever the destination stagename is known to the Messenger, the message is placed in the appropriate Cast member's mailbox. Otherwise, it is passed to the Executive Director's Messenger and routed to the appropriate Host.

There is one Executive Director in SIMULACT which coordinates all Cast member activities over an entire network. The Executive Director provides the link between Directors necessary for inter-machine communications, directs each Grip so that synchronization throughout the network is maintained, and handles the interface between the user and SIMULACT.

4.4.1.3 Concurrency Control In SIMULACT Concurrent execution of n Actors on k machines ($n > k$) is emulated through the imposition of a *time frame* structure in execution. A time frame cycle breaks down to three fundamental parts: invocation of the Ghosts, the distribution of mail by the Messengers, and invocation of the Actors. For SIMULACT distributed over two hosts, Figure 6 depicts a representation of two time frames.

At the start of the first time frame, the Executive Director notifies both Directors to begin executing Ghosts. (This models the occurrence of events in the world external to the distributed system.) At the conclusion of the Ghost frame, each Director automatically invokes its Messenger. The Messenger distributes all messages which were generated during the current Ghost frame, as well as all those resulting from the previous Actor frame. Mail destined for Cast members residing on the same host processor is placed in the appropriate mailboxes. The solid line extending from each Director's Messenger represents the transfer of non-local mail to the Executive Director's Messenger. In order to reduce network overhead, this transfer is done in

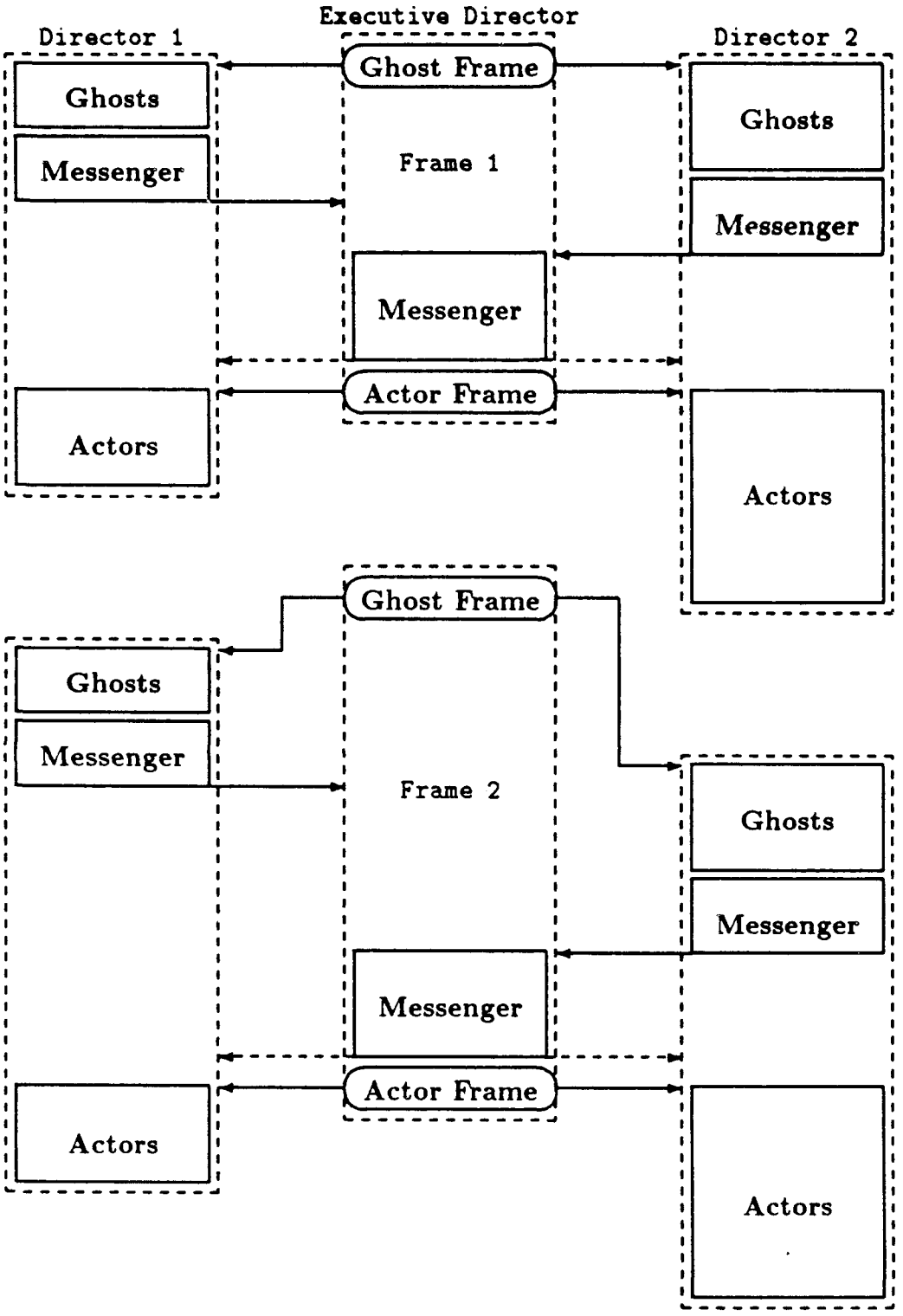


Figure 6: Activity in SIMULACT

the form of a single message. This communication always occurs, even if there are no messages to distribute, as a synchronizing mechanism for the time frame so that Actors cannot "run away". After sending this message, each Director enters a wait state until the Actor frame directive is received from the Executive Director. The dotted line directed out of the Executive Director's Messenger represents the possible distribution of inter-machine mail prior to sending the Actor frame instruction. The Executive Director's Messenger is invoked immediately following the receipt of the last Director's Messenger communication.

Upon receiving an Actor frame command from the Executive Director, the Director's Messenger is invoked to distribute any inter-machine messages that may have been received. Next, each Actor is allowed to run for one time slice (time frame). At this point the Executive Director immediately enters its next time frame cycle, sends the Ghost frame command, and waits for all the Director Messengers to send their next synchronizing signal. Again in the second time frame of Figure 6, it is Director 2 which requires the most time to run.

4.4.1.4 User Interface Facilities There are five user interface facilities that will be discussed in this section. These facilities provide mechanisms for inter-agent communication (Mail), code sharing (Support Packages), interactive monitoring and debugging (Peek and Poke), post mortem trace analysis (Diary), and runtime monitoring (Gauge). These features were designed to make SIMULACT more attractive as a development environment for expert systems.

Depending on the constraints and characteristics of the expert system being developed, the application programmer constructs a network environment of intelligent agents which collectively work together towards the satisfaction of one or more goals. SIMULACT provides several mechanisms allowing these agents to communicate without being concerned with implementation details.

In general, communication between agents occurs when one agent sends a packet of information to another, addressing the target agent by its stagename. The format of these packets is not specified by SIMULACT. Instead, it is left up to the user to formulate a syntax that is convenient in the context of the system being developed.

The **send-memo** function is the simplest mechanism one Cast member can use to communicate with another. This function accepts two arguments: the stagename of the destination Cast member and the memo to be sent. Automatically, at the beginning of the next time frame, this message will appear in the destination agent's mailbox. Each memo contains the stagename of the sending member, as well as a timetag indicating when it was sent. It is the responsibility of each Cast member to periodically to check its mailbox for incoming messages.

Many communications between agents take the form of requests for information.

Using the send-memo function requires that a sending agent sort its mail to retrieve the reply to a request after it has been received. Futures [11, 22, 40, 42] provide an agent with a mechanism for sending a message that returns a result. A memo sent using the future facility in SIMULACT returns a data structure called a future. After the memo has been received and processed, the result is routed back to this data structure. The sending agent uses this future to determine when the result is available, and to extract it after it has arrived. SIMULACT provides this capability through the **send-future** function.

In some cases, requests for information may not have one definitive reply. Instead, pieces of information may be returned at different times. SIMULACT allows two Cast members to establish a *future stream* between themselves for returning results over time. The user specifies criteria for determining when a future stream should be closed.

A Support Package contains code that can be accessed by several Cast members, thus reducing memory requirements. As in any shared memory system, an integrity violation could occur whenever a Support Package accesses or alters global information. The underlying assumption concerning independent environments for each Cast member would be violated. To guard against these problems, SIMULACT detects the potential occurrence of integrity violations and warns the user when a Support Package tries to instantiate a global variable. Ideally, Support Packages should contain purely functional code. However, this restriction would severely constrain the code that can be placed into Support Packages.

There are two ways to use Support Packages other than for purely functional code. One way is for a Cast member to pass a local data structure as an argument to a Support Package function. If that function is "for effect", the result could then be bound appropriately. The other method requires the application programmer to use SIMULACT's **sim-set** functions. Basically, the sim-set function allows the Support Package to alter a global variable that is present in each of the Cast packages. The goal of the Support Package facility is to reduce overhead. Use of Support Packages does reduce the overhead, but it does so at the expense of requiring that the user have more knowledge about SIMULACT's implementation and Lisp packages [45] than might be desirable.

This facility can be invoked at any time when running an expert system as a monitoring and debugging tool. It allows the user to enter the local environment of any Cast member and to examine or change any part of its environment. The Peek and Poke is invoked through a menu and displayed at the Executive Director's host. However, any Cast member residing on any host can be accessed.

The Diary facility can be used as a debugging tool, or simply as a mechanism for post mortem analysis of system behavior. There are three levels of Diaries, all of which may be independently active. The Executive Director's Diary, when turned

on, records all inter-machine messages handled. Director Diaries record all local communications, while Cast member Diaries record all screen activities. All Diaries are written to files to permit post mortem examination.

This facility is used by SIMULACT to display the current elapsed run time, current mode of operation, and the *time frame ratio* which is a measure of SIMULACT's distributed performance. It can also be used as a runtime monitoring device by the user. The **make-gauge** function accepts two arguments: a string to be displayed in SIMULACT's gauge window and a function to be evaluated periodically. SIMULACT automatically evaluates this function and updates the gauge window appropriately throughout the execution of the expert system.

4.4.1.5 Performance Issues The overhead incurred in managing the emulation of a distributed environment is one important measure of system performance. In this section we present preliminary results indicating the overhead incurred by SIMULACT as implemented on one and two machines. We first outline the experiments which were performed, then discuss the results obtained.

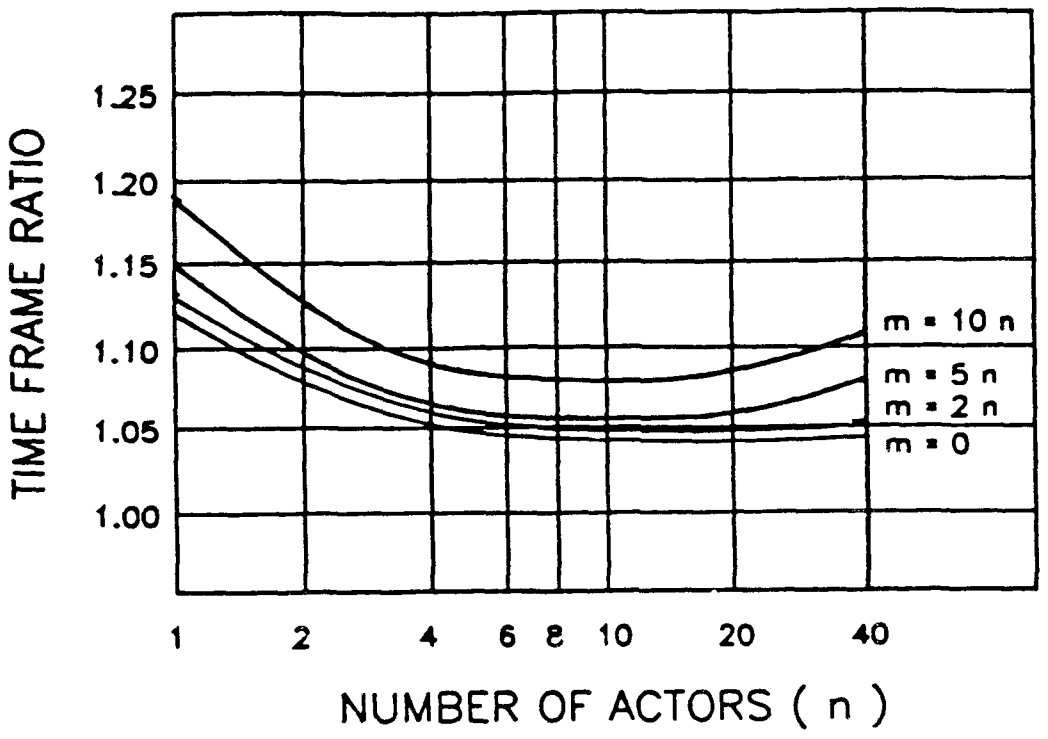
Our experiments were designed to obtain results that would assess SIMULACT's behavior as the number of messages per time frame increases. In each of the experiments, the number of messages per time frame, m, was varied over the range 0 to $10n$, where n is the number of Actors in the system. Each Actor process worked continually, consuming its total time slice allowed per time frame. Thus when $m = 0$, we measured SIMULACT's best case performance. It should be pointed out that in the distributed case where $n > 0$, the number of messages per time frame in our experiments represented entirely inter-machine communications, emulating a worst case scenario.

The measurement used to represent SIMULACT's performance was a *time frame ratio* gauge. This ratio is defined as:

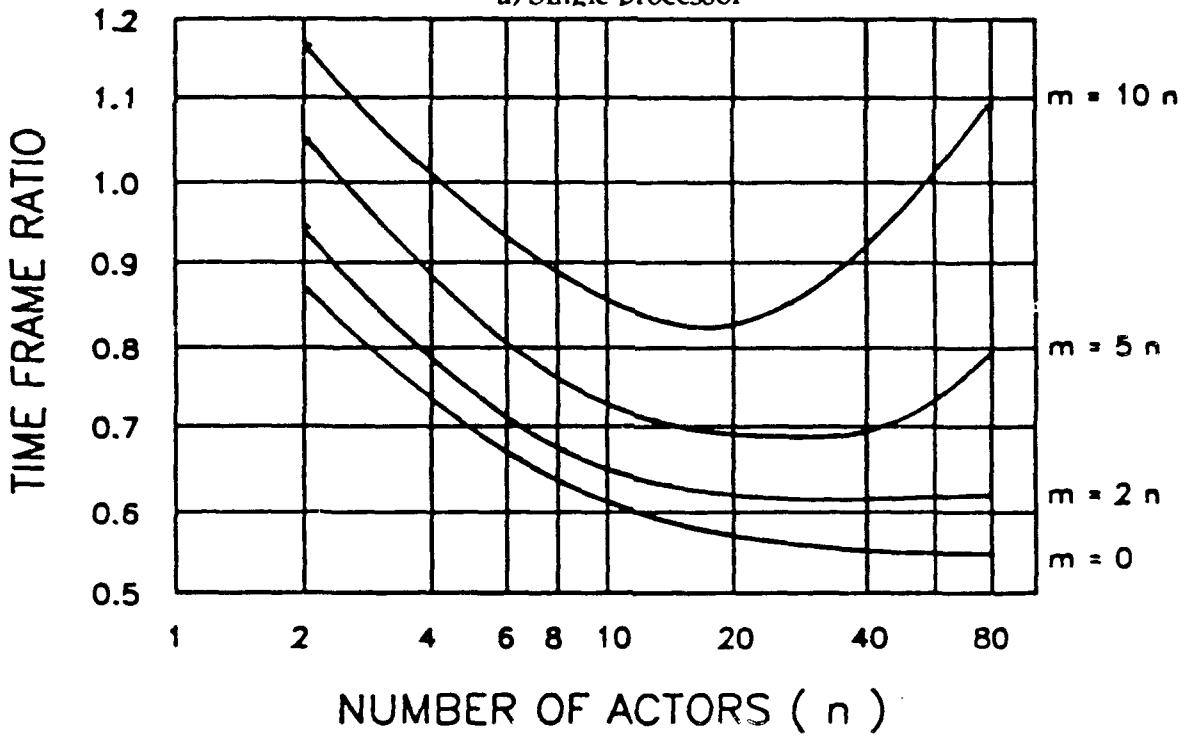
$$\frac{\text{elapsed wall time}}{\text{sum of all Actor elapsed time}}$$

This ratio times the number of Actors in the system provides an estimate of how much time is required by SIMULACT to execute one time frame. For the ideal situation involving no overhead, this ratio would be 1.0 and 0.5 for the one and two machine cases respectively.

Figure 7 a depicts our experimental results for SIMULACT running on a single processor. Figure 7 b shows the two processor results. In each case, data was collected over a range of 1 to 40 Actors per processor and a one second time frame was specified. These results are preliminary and modifications to SIMULACT have already been



a) Single processor



b) Two processors

Figure 7: SIMULACT's performance

scheduled to enhance performance. Further testing is planned to observe the effect of three machine distribution as well as varying time frame sizes.

For the single machine case with no message passing, SIMULACT's overhead approaches 4.5%. Similarly, the distributed case approaches 10% overhead. Both sets of curves indicate that as the number of messages per time frame increases, so does the overhead. In fact, between 100 and 200 messages per time frame handled by the system seems to be a saturation point for the Messenger. Currently the Messenger uses an a-list to associate stagenames with Cast members. We should see improvement when this is implemented as a hash table lookup. Also note that the distributed case after saturation degrades at a much faster rate. One explanation for this can be deduced from Figure 2. All inter-machine messages are handled three times by different Messengers and must be sent over the Lisp machine network. Messages among agents residing at the same host processor are handled once by the Messenger and sent directly to the appropriate mailbox.

4.4.1.6 Status We have described SIMULACT, an environment for the design and development of distributed intelligent systems. SIMULACT is written in extended Common Lisp, and is currently running on a network of Symbolics 3670 and TI EXPLORER Lisp machines. Our implementation makes extensive use of flavors to improve data encapsulation and to facilitate the modeling of environments in which a group of semiautonomous processes do not share common memory.

SIMULACT has been particularly useful in the development of a distributed planning system [7]. It has been used to expose the nature of message traffic in this planner and to develop and debug plan generation in a distributed environment. SIMULACT has also been used as an aid in the development of a distributed theorem prover [25]. In each of these projects, SIMULACT's modularity and transparency have allowed us to concentrate our efforts on the development of these agents rather than on the problems associated with managing a distributed environment.

4.4.2 Graphical User Interface for Structural Knowledge (GUS)

Knowledge acquisition is a crucial phase in constructing knowledge-based systems. This process consists of identifying, formalizing, and representing the relevant knowledge. Application domains which involve reasoning about physical systems usually include knowledge about the structure of the target system. For this reason, we have designed and implemented a tool to assist an expert in conveying structural knowledge in a form suitable for machine manipulation. Although the present design is directed toward a specific application domain, the design principles employed are domain independent.

When a knowledge engineer questions an expert about problem solving for some

physical system, the expert will often begin with a sketch of system components and their interconnections. The symbols used by the expert to represent components and interconnections comprise a language for structural knowledge description. Verbal reasoning and explanation about the behavior of the physical system proceed, with the expert using the sketch as an aid in his or her description. It seems clear that a diagram of a physical system often embodies what is known about structure. This knowledge is represented using a set of graphical symbols or icons that are specific to the domain of interest. For this reason, a graphical interface for capturing structural knowledge should be built upon the symbols used by the expert for structural knowledge description. Furthermore, the composite diagram of domain specific symbols can then be used as an explanation facility, much as it would in a non-automated environment. Part of the gap between expert and machine is bridged by providing a common language.

The graphical user interface system we developed enables an expert to easily draw a diagram of a physical system with a graphical interface tool, automates the machine extraction and interpretation of embedded structural knowledge from the diagram, and forms the machine representation of interpreted knowledge.

4.4.2.1 Overview We have developed a Graphical User interface for Structural knowledge (GUS) which provides an interactive, mouse and menu-driven interface for capturing the structural knowledge for a specific application domain: large scale communications network systems. Our implementation of GUS is currently running on a Symbolics 3670 Lisp Machine and is written in Zetalisp [24]. A combination of the mouse, menus, window system, and object-oriented flavors package provided the necessary tools for building GUS. User interaction is primarily via manipulating a mouse-controlled cursor. Components are selected with the mouse for addition from a library of component icons and then positioned upon the drawing area. Connecting components follows a similar pattern: select the type of connection desired from the library of connection icons and select a component and connect it to another component in the drawing area. Attribute values for objects are easily edited via menus. Continuation of this process results in a complete graphical display representing a communications network with specific equipment configurations. Additionally, a knowledge base which embodies the captured structural knowledge is constructed.

The key idea here is the following: knowledge base building is dynamic and transparent. Knowledge base building is dynamic in the sense that editing of the graphical model also edits the model's knowledge base. Knowledge base building is transparent in that the user need not be concerned about how objects are represented, only how displayed component objects are graphically positioned and connected to comprise a composite communications network model.

4.4.2.2 Design Objectives Our design of GUS reflects three basic criteria for a knowledge representation language. The first is expressive power. How easily can the expert communicate his knowledge to the system? Supporting this criterion is the extensive use of domain specific icons representing components and connections. These icons form a natural vocabulary of symbols which are the foundation of a language for structural knowledge description. The second important criterion is understandability. Can experts understand what the system knows? Machine captured structural knowledge is represented graphically with the same component and connection icons used by the expert to convey structural knowledge to the machine. This commonality of structural knowledge expression supports an environment for natural comprehension of machine knowledge. The final criterion is accessibility [19]. Can the system use the knowledge it has captured from the expert? From a *system* perspective, the purpose of this interface is to create a system knowledge base consisting, in part, of structural knowledge. Simulation techniques and problem solving agents such as fault isolation, service restoral and performance assessment make heavy use of structural knowledge represented in the system knowledge base.

4.4.2.2.1 Conceptual Design Perspective An object-oriented design methodology was adopted for design and implementation. An object is the primitive element of an object-oriented programming environment. Larger or more complex objects are formed from the composition of simpler objects. What is interesting is that objects combine data and functions. Data is represented in attribute slots of the object and functions which utilize this data are resident in attached structures called methods. Communication with an object requires a message to be sent to the object. The function contained by a method is executed in response to receipt of a message by the object. A method capable of receiving this message must be associated with the receiving object. The result of function execution can be the alteration of object state (modification of attribute slot values) or the execution of a predefined task.

Object-oriented programming has been used in many systems to create interactive, menu-driven graphical applications [31]. This is the result of application entities (graphical icons, menus, mouse sensitive text) being naturally represented by objects. Structural representation of many physical application domains is greatly facilitated by objects. For example, objects of an electrical circuit application domain would include resistors, capacitors and transistors. Methods for a resistor could include **calculate-your-voltage-drop** and **short-yourself-out**. Thus, by sending the **calculate-your-voltage-drop** message to a resistor object, the associated method utilizes object data to determine voltage drop. The effect of a resistor may be removed by sending the **short-yourself-out** message (this could be implemented by altering object state knowledge, i.e. setting the resistance attribute to 0). Expansion along these lines will result in a powerful representation of physical circuit domain objects useful for circuit fault diagnosis techniques. Paralleling these ideas to other physical

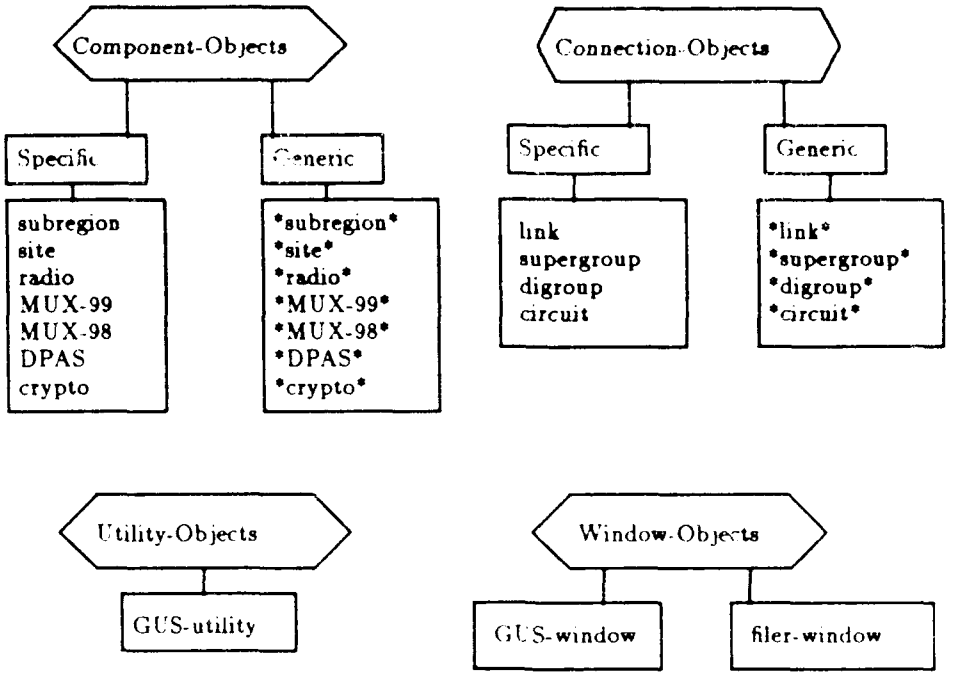


Figure 8: Object Type Classifications and Objects

application domains is a natural extension. Consequently, elements of interest in our application domain are modeled by objects.

The first step in the object-oriented design was the specification of type classifications of objects. To conform to the structural theme of representation, four type classifications were conceived: component-objects, connection-objects, utility-objects and window-objects (Refer to Figure 8). Component-objects contain those objects which model components of the application domain. Connection-objects is a collection of objects representing different types of connection media. Utility-objects consists of one object which is responsible for all information regarding graphical input and output. Window-objects contains the window objects used for graphical editors and the file system interface.

4.4.2.2.2 Functional Design Perspective Three fundamental functions are provided by GUS. First, structural knowledge is captured from an expert. Second, this knowledge is interpreted and represented in the structural knowledge base. Third, this knowledge is displayed graphically (Refer to Figure 9).

Knowledge about the structure of our application domain is captured via interpretation of graphical input. Graphical input is performed by mousing component

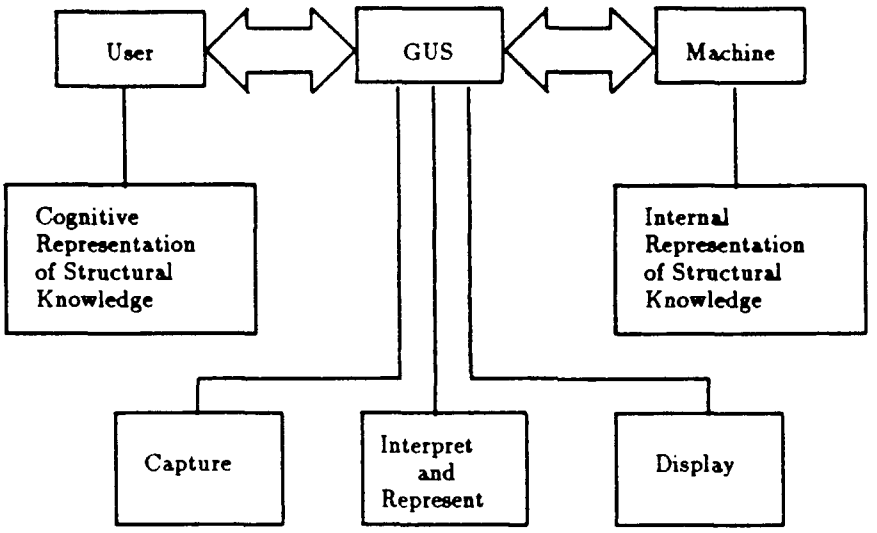


Figure 9: Fundamental Functions Provided by GUS

icons and adding them to a configuration and selecting connection icons in order to connect added components. As the user draws a component or a connection with the mouse, the machine also interprets the graphical addition of this component or connection as an addition of the internal representation of this component or connection to the knowledge base. As discussed earlier, the internal representation used is an object. Therefore, graphical addition of components and connections results in the addition of instances of objects representing these components and connections to the knowledge base.

Inherent to equipment connectivity are structural constraints. Depending upon the type of connection, there are certain valid endpoints. Therefore, a functional agent tightly coupled with capturing structural knowledge is necessary to guide the user in selecting valid endpoints in the context of the type of connection being added. With the aid of dynamic mouse sensitivity and highlighting techniques, these connection constraints are effectively enforced.

Upon the completion of a graphical configuration, a knowledge base representing the configuration is also completed. This knowledge base contains component and connection objects, all of which are related in some physical sense. Two types of knowledge are represented in the knowledge base: graphical and state knowledge. Knowledge concerning graphical display encompasses all information necessary for graphical representation of an object. State knowledge embodies specific and generic attributes of the object. Since a frame-based knowledge representation is employed, both types of knowledge are stored as attribute slot values of appropriate objects.

The last function provided by GUS is the display of structural knowledge. Configurations drawn on the computer screen can be saved and loaded later as needed. The display of machine represented knowledge is comprised of the same component and connection icons used in the drawing of the configuration when it was saved. Each component and connection icon is a representation of a component and connection in the knowledge base, respectively.

4.4.2.3 Design Implementation This section gives more detail about object classes. Additionally, input and output techniques, mouse sensitivity and structural hierarchy from an implementation point of view are presented.

As discussed in Section 4.4.2.2.1, the first step in an object-oriented design was the specification of object type classifications. The next design step is the specification of object types within each classification. Object types of component-objects are site, radio, MUX-99, MUX-98, DPAS, and crypto. Types of connection-objects are link, supergroup, digroup, jumper and circuit.

The single object within the utility-objects classification is GUS-utility. The GUS-utility object is the heart of the interface and coordinates graphical display of component-objects and connection-objects.

The window-object classification has two object types: GUS-window and filer-window. GUS-window is a customized window for graphical input and output and is the basis upon which the network and equipment editors are built. User interaction is primarily supported by a level specific library of icons. That is, for each level of structural detail, there are associated icons which form the basis of user interaction. Filer-window provides an interface window for binary file saving and loading of network configurations.

The input device primarily employed by the interface design is the mouse. The advantage of user input based upon pointing rather than typing in a command is that seeing something and pointing to it is significantly easier than typing. From a psychological viewpoint this issue is known as recognition versus recall. Numerous experiments based on distance, target size, and learning found the mouse fastest and with the lowest error rate relative to other input devices such as joysticks and keyboards [34].

Graphical input by the mouse is chiefly supported by two drawing techniques. The first technique is an implementation of a library of items to be displayed as icons on the screen for convenient selection and placement in the drawing. There are two reasons for the use of icons. First, icons are visually more distinctive than a set of words. Second, an icon is able to represent more information than words in a small place, and conservation of screen space is of high priority for items not directly part of a drawing [34].

The second drawing technique implemented is rubber-banding. This technique is utilized during the placement of a connection. It allows the user to strategically place a connection and see what it will look like before fixing it in place. An additional capability provided as part of the connecting process is *tacking down*. Tacking permits the user to tack a connection down at specified intermediate points (as opposed to endpoints). This capability enables the user to specify connections other than point-to-point straight lines; specifically, connections comprised of line segments. Therefore, connections can always be specified such that any one segment of any given connection is parallel or perpendicular to other existing connections.

A graphical coding technique is used for graphical output. Icons are used to represent physical component and connection objects of the application domain. The symbols used for component and connection icons of the icon library are the same symbols used for graphical output. For instance, the addition of a radio to an equipment configuration results in the output of a radio symbol to the screen. This radio symbol represents the newly created radio object added to the system knowledge base. This radio symbol is also the same symbol used to comprise the radio icon. Hence, the graphical coding technique stems from the idea of each displayed graphical symbol not only being a visual display, but also a representation of a physical object of the application domain and an object in the knowledge base.

A common source of erroneous input in a menu-driven graphical interface is using the mouse to select a menu command or displayed object when such an action is out of the current context. For example, choosing to add a connection or remove a component and having all object types displayed be mouse sensitive would be of poor design. A solution to such problems is dynamic control of mouse sensitivity to implement the concept of *context sensitive* mouse sensitivity.

By dynamically controlling the mouse sensitivity of displayed objects, the mouse is context sensitive in the sense that the items which may be pointed at with the mouse are dependent upon the current context. For example, when an icon command representing a component or connection is selected, a menu of commands will appear. These commands are only associated with the type of object represented by the icon. Component icon commands for *type* component only provide mouse sensitivity for *type* components. Similarly, the *type* connection icon addition command only provides mouse sensitivity for those component objects which are valid *type* connection endpoints.

Multi-level structural knowledge is an inherent property of the physical structure of the application domain. In the following, a representational view of multi-level structural knowledge is described. The connectivity of a communications system can be edited at all three levels of structural detail, but only displayed at two (the network and equipment levels). The ability to represent and edit all levels of structural knowledge was a principle design objective.

At the network level, site connectivity is represented by sites and respective link interconnections (See Figure 10). A library including icon symbols representing sites, links, and subregions have associated with them command menus. By selecting the appropriate icons and subsequent commands, the user assembles a drawing representing site-to-site connectivity of a network. Once the site connectivity has been specified, editing of either the subregion or equipment level of structural detail is permitted.

Subregion editing, the most abstracted level of structural detail, is achieved by the appropriate selection of subregion icon commands. Grouping sites together and designating a control center comprises a subregion and consequently, a new subregion object is added to the system knowledge base.

From the network level, selection of the EQUIPMENT EDITOR icon command and subsequent selection of a site brings the user to the equipment level (See Figure 11). At this most detailed level of structure, internal equipment editing and connectivity of a site can be specified. A library of equipment icons, similar in design to the library of network level icons, encapsulates configuration commands. While it is true that the equipment level represents the equipment configuration at a particular site, it is important to remember that equipment configuration is, in a sense, continuous between sites. That is, links specified at this level of detail are representations (pseudo-links) of links existing at the network level. A link at the network level may only be represented at the equipment level if it is connected to the site at which equipment configuration is taking place.

4.4.2.4 GUS Architectural Design There are several important features of the design which are discussed in this subsection. First, GUS has two basic editors, the network editor (for specification of site and subregion connectivity) and the equipment editor (for specification of equipment connectivity). Second, operations are grouped into two categories: components and connections. This decision was made because there are number of logical similarities among operations. This grouping of operations provides an environment in which the user may interact with the system in the same way for all operations in a given category. Similarly, physical objects of the application domain are represented by two groups of objects: component-objects and connection-objects. Third, a variety of connection constraints are enforced during connection specification processes. Each constraint is enforced in a consistent manner by using dynamic mouse sensitivity. Guidance is provided to the user in the form of highlighting to show which displayed objects are mouse sensitive.

Mouse sensitive objects are those displayed objects which react in a controlled manner to the positioning of the mouse cursor over them. Certain terms derived from the word *mouse* are commonly used in the realm of mouse pointing devices and their application. For instance, *mousing* refers to selecting with the mouse, *moused*

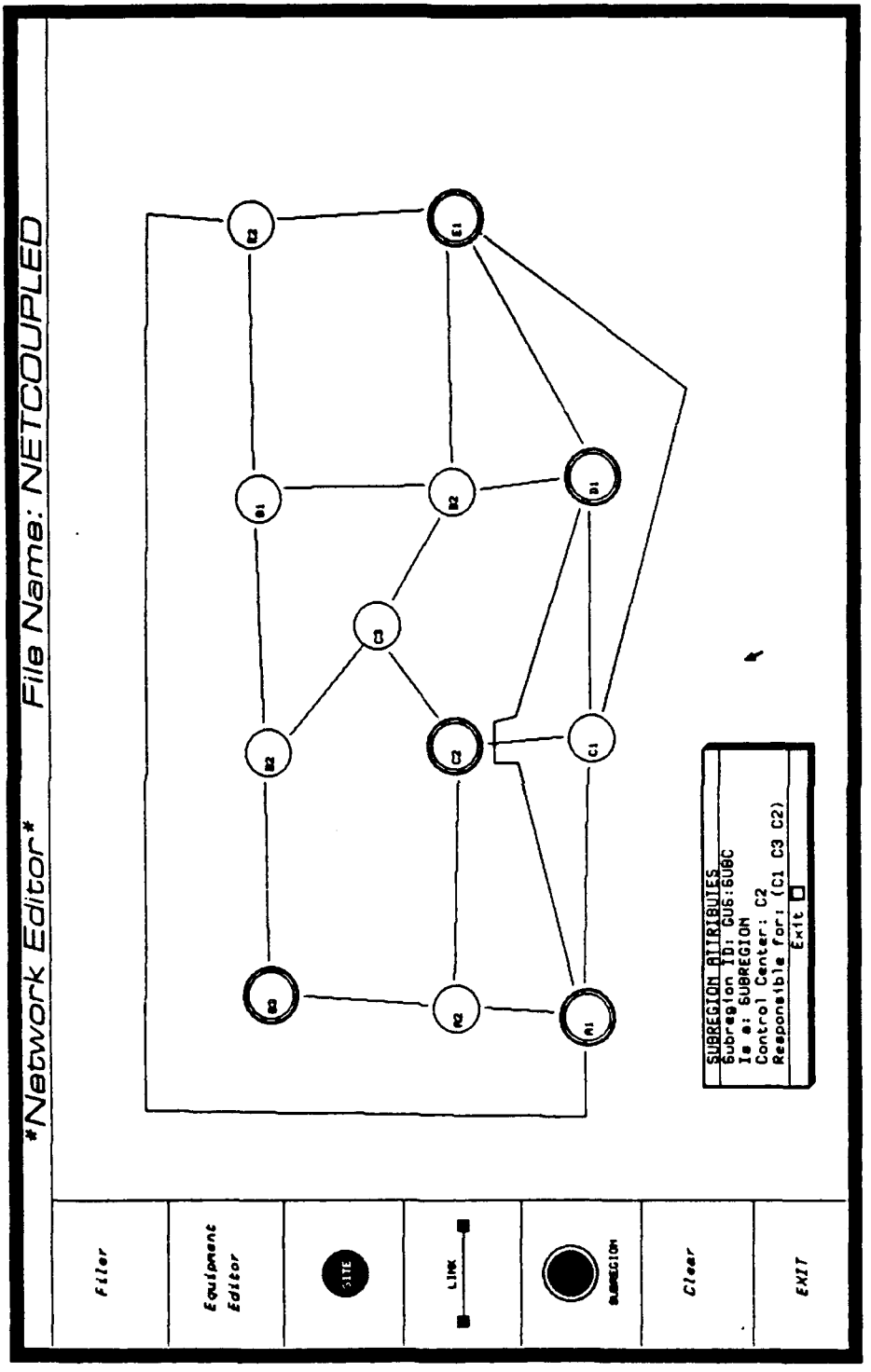


Figure 10: Example Network Editor Screen

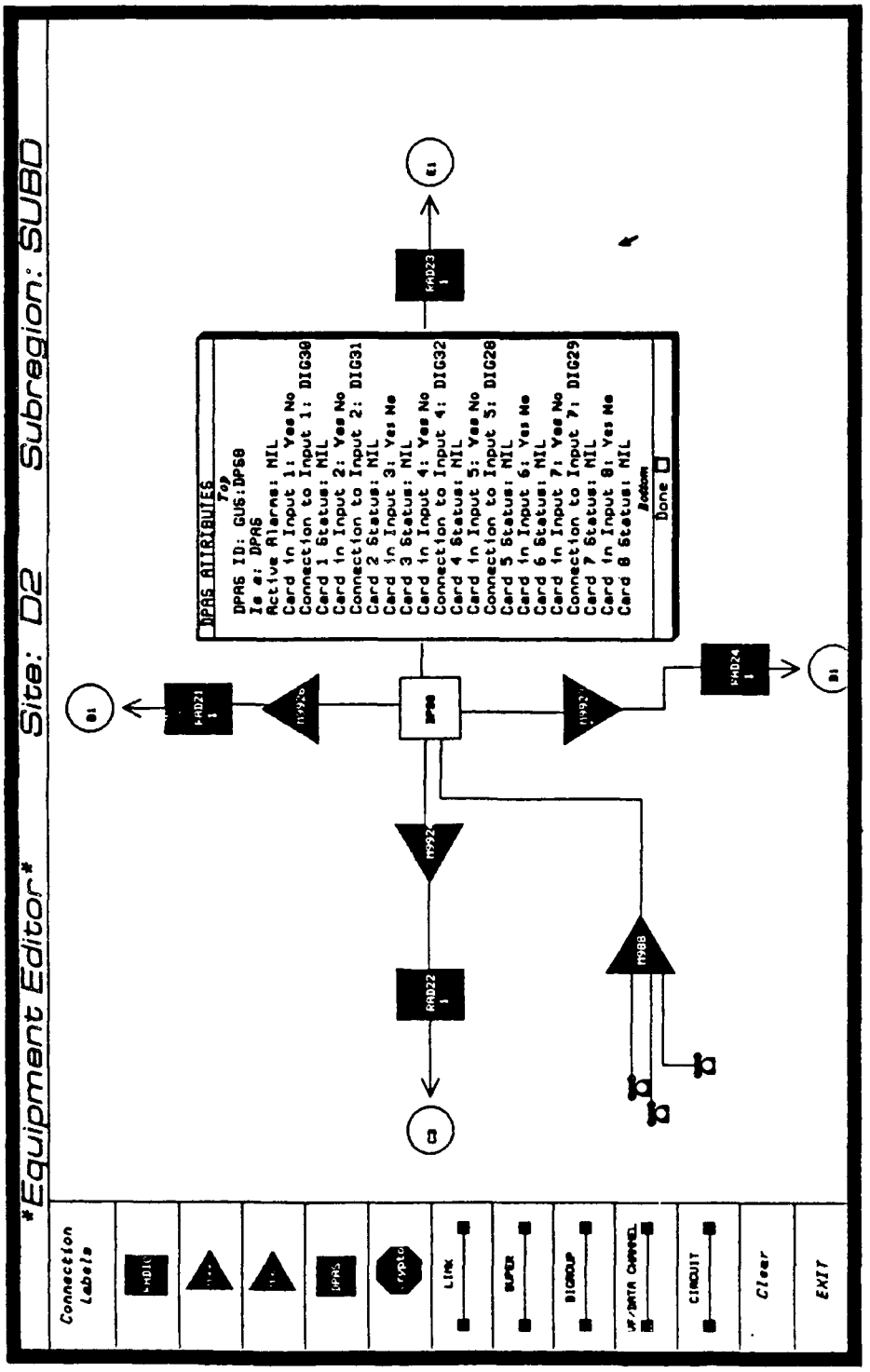


Figure 11: Example Equipment Editor Screen

means *selected*, and *mouseable* means the capability to react to the mouse.

4.4.2.4.1 User Interaction User interaction with GUS is primarily through mouse and menu driven commands. Mouseable graphic icons and commands in each editor represent groups of type specific sub-commands for the type of object represented by the icon or command. From a graphical perspective, those graphical icons which represent objects that are components of the network model are the same graphical displays used by the graphics support when adding a component to a configuration. This visual one-to-one correspondence is the foundation of our symbolic language for structural knowledge description.

User interaction is implemented by means of two physical devices, the keyboard and the mouse. Keyboard input is limited to situations in which the user must supply data values that cannot be predicted or guessed by the system. The majority of user input is via the mouse, with the use of mouseable commands and graphic icons. Most often the selection of a mouseable command or graphic icon results in presentation of more commands in menu form. The use of menus and pointing devices is a preferred implementation of user interaction because the user is presented a set of possible command choices (dependent upon the current context) rather than being required to remember commands. Additionally, only context dependent commands are available for choice, consequently, erroneous command choices are avoided.

Pop-up menus are used in GUS force user input by remaining displayed until a choice has been made. This type of menu is typically used for selection of available links (link addition at the equipment level) or inputs (choosing a host input number for a connection to a piece of equipment). The item choices displayed at a particular time in one of these menus are the result of some evaluating processes. For example, the input menu only displays spare input numbers of a piece of equipment (those inputs which do not already host a connection).

4.4.2.4.2 Network and Equipment Level Design As discussed earlier, our application domain exhibits a high degree of hierarchical structure. GUS captures structural knowledge at two levels of structural detail (the network and equipment levels) via the network editor and the equipment editor. The editing of structure in either editor is limited only by constraints imposed by mouse state diagrams and an implied ordering of data input. In this way, the user is given a freedom of input which is bounded by mouse state.

The network editor is an icon-based menu-driven user interface for the creation, editing and saving of network level components, connections and subregion designations. With simple mouse and menu commands the user can add sites and make link connections by selecting sites as endpoints. The equipment level is similar in design

to the network level. Entering the equipment editor places the user in a familiar environment.

The equipment editor is an icon-based menu-driven user interface for the editing of equipment level components and connections. It provides an easy and natural means of configuring equipment within a site location and is accessible only through the network editor by selecting a site at which equipment configuration is desired.

One point of interest, from a human factors perspective, is the specification of connections between equipment. Depending upon the type of connection, certain constraints regarding endpoints must be observed. This was not the case at the network level since there was only one type of connection and one type of component. The only connectivity constraint was the common sense constraint of a link not having the same site for both endpoints.

4.4.2.4.3 Enforcing Connection Constraints Types of connections found at the equipment level are links, supergroups, digroups, jumpers, and circuits. As we have mentioned, there are endpoint constraints for each type of connection. In order to enforce connection constraints during the connection specification process, a combination of dynamic mouse sensitivity and graphical highlighting is employed. Highlighting valid choices makes it easy for the user to identify mouse sensitive components. For any connection type the user selects, only those pieces of equipment which are valid endpoints are highlighted and mouse sensitive. Which endpoints are valid is dependent upon the type of connection, which endpoint is being specified, and the presence of spare inputs or outputs to host the connection.

In order to provide design guidelines for connection specification, two general constraints are imposed on the user during the connection specification process. First, a connection will always start at a piece of equipment. Second, if both endpoints of a connection are constrained to be equipment, then the first endpoint will always start at the piece of equipment which hosts the connection as an input.

Link addition or removal at the equipment level does not actually add or remove a link to the knowledge base. Instead, equipment level links can be thought of conceptually as *pseudo-links* or representations of links at the network level. The network level provides abstracted information concerning network connectivity that consists of link connectivity for all sites, whereas the equipment level provides connectivity information only for the site at which equipment configuration is taking place. Additionally, the equipment level provides information consisting of which radios each link is connected to and complete equipment connectivity at that particular site. Consequently, link connections hold the special status in that they are the only type of connection which bridges the network and equipment levels of representation.

Several link connection constraints must be satisfied. A site can only have as

many links specified in its equipment configuration as there are links connected to it in the network configuration. Link connections (at the equipment level) always start at a radio and end in free space. Only one link may be connected to a radio. Note that there may be more radios than links at the network level, but there will remain extra radios which are not connected to any links. Consequently, these radios would not be part of the equipment connectivity and would constitute an incomplete representation of a network system.

A special type of connectivity specification is supported for configuring DPASes. DPASes only host digroup connections. The function of a DPAS is to interconnect digroups at the channel level of connectivity. DPAS configuration is supported by a DPAS configuration editor window. This window is physically comprised of an input completion pane at the top of the window, with the remaining bottom three-fourths divided column-wise into an input pane on the left and a configuration pane on the right.

4.4.2.4.4 Circuit Representation A circuit is a complete path (consisting of connections and equipment) from one MUX-98's input to another MUX-98's input. There is at least one level of multiplexing involved in each circuit. Circuit specification is easily performed by the user, though it has been a very complex task from a design and development perspective.

The addition of circuits should only take place after all sites have been configured at the equipment level. This is because circuit addition requires a complete path from the originating MUX-98 to the destination MUX-98. If an incomplete path is discovered during the path seeking algorithm, then the addition process for that particular circuit is unconditionally aborted and information about the problem area is presented to the user. One side benefit of circuit specification is thus knowledge base consistency checking. Successful completion of the circuit path seeking algorithm indicates that the specified equipment connectivity for that circuit's path is logically sound and meets connectivity constraints imposed upon circuit paths.

4.4.2.5 Comparison With Existing Tools An existing knowledge representation tool in use today is Intellicorp's KEE (Knowledge Engineering Environment) system. KEE offers many graphics tools, some of which have many conceptual and functional characteristics similar to those of the graphical interface we have developed.

The KEE system is a development environment for building models and reasoning about and analyzing those models. Within the KEE environment there are graphics tools which help users construct graphic images, image libraries, and interfaces via an object-oriented implementation. Thus, KEE has a frame-based knowledge base consisting of objects and their associated attribute slots. Some of these tools include

KEEpictures, ActiveImages, and SimKit. KEEpictures assists the user in constructing customized, graphic images and interfaces. ActiveImages is a library of images constructed with KEEpictures. Of particular interest is the tool SimKit. With SimKit and a library of graphical simulation objects, non-programmers can easily build, run and modify simulations with simple mouse-and-menu commands.

The interesting aspect of SimKit, for our purposes, is not its simulation abilities, but the mechanism and procedure by which models are built and represented. Users are able to interact with the application by manipulating images with a mouse-controlled cursor. A library of icons representing simulation components is used to build complete simulation models. As components are selected from the library's menu of icons for addition to the simulation model, new members of the class of the simulation component represented by the icon are automatically created and added to the model's knowledge base. Attribute slots are utilized to represent the modeled objects' attributes and their corresponding values. Additionally, connections between component models are represented by slots. The salient feature here is that the explicit addition of objects to the knowledge base is avoided by having knowledge base modification be a consequence of graphical editing with the mouse.

In GUS, user interaction is primarily through mouse-controlled manipulation of a library of domain specific icons. These icons represent the components and connections of the application domain. As components are selected from the library of icons for addition, new instances of objects represented by the icon are automatically created and added to the system knowledge base. This is a concept held in common with SimKit. A frame-based knowledge base implementation is also utilized with slots and default values representing modeled physical object attributes. Another major commonality is that knowledge base building is completed implicitly by the addition of component and connection icons to comprise the physical architecture of the target system.

A limitation of our graphical interface is that it is domain specific. The library of domain specific icons is fixed and not modifiable via the interface tool itself. However, this is not a limitation of the graphics capabilities of SimKit. SimKit permits loading in of a library of icons. Custom application libraries can be created with the use of KEEpictures and ActiveImages.

A design goal of our interface, which is not apparent in SimKit, is to have knowledge concerning graphical display of a modeled physical object be loosely coupled to structural knowledge of the system. Although implementation and the extent to which SimKit addresses this goal is unclear, it is believed that our approach is unique. A common approach to coupling graphical display capabilities is via inheritance by mixing in graphical display objects to objects of the application domain. Our approach does not follow this conventional technique, but instead allows the structural knowledge to be represented completely independently from the graphical display

knowledge.

4.4.2.6 Status The intent of this research effort was to develop a graphical interface tool for the construction of a structural knowledge base representing domain specific knowledge for a communications network system. In its current implementation, GUS is proving itself to be an effective tool for knowledge base construction. Design criteria and objectives have been satisfied and in some cases exceeded.

Although the present design was directed toward a specific application domain, the design principles employed are domain independent. The motivation behind the development of a generic interface tool for capturing and representing structural knowledge of a variety of application domains is obvious. The careful attention paid to modularity in designing GUS has resulted in a system which can potentially serve as a prototype for a generic interface tool. Specifically, the use of an objected-oriented approach lends itself to a domain independent extension by allowing the possibility of user-defined object classes. A library of domain specific objects and their attributes could be created and represented. These objects would then be associated with predefined object classes which have generic operation capabilities. In this way, domain specific objects are created and acquire operational capabilities (from an interface perspective) by being associated with a predefined object class. The creation of mouse sensitive regions for domain specific objects and the creation and association of graphic icons and textual commands to these mouse sensitive regions must also be supported. User creation of certain menu types with user-specified items should also be supported.

A limitation encountered with this implementation of GUS is that the window size constrains the size and complexity of a represented network system at both the network and equipment levels. This is a consequence of component objects having a fixed location for display. Investigation of zooming and panning techniques and their potential application to this specific problem would be an excellent approach. At present, absolute screen pixel coordinates are used for representation and display of objects. Zooming could be implemented by using a window-relative representation and a scaling technique on absolute screen coordinates for display. Implementation of panning follows from the use of relative coordinates for zooming. Depending upon the current scale, displayed object coordinates would be relative to a scaled "home" pixel coordinate. Various regions of the display could then be viewed by detection of mouse cursor movement in editor window margins (similar to scrolling window capabilities). Objects displayed would be displayed relative to the home coordinate whether it is visible or not. The addition of zooming and panning capabilities permits the display of network systems with real world longitude and latitude locations and realistic proportions. Thus, the representation and modeling of existing network systems and the creation and modeling of hypothetical network systems closely related to real

world coordinates would be a salient characteristic of the interface tool.

Another limitation is configuration completeness. Although consistency checking is provided by connection constraints, it is still possible for a user to construct an incorrect configuration from a completeness perspective. Incomplete configurations are detected when specifying circuits and indicate to the user that the current network structure and corresponding knowledge base must be modified.

Future research efforts could address some of these limitations and proposed solutions. Special emphasis should be applied to the extension of the design of GUS to a generic interface tool. From this perspective, GUS is a valuable prototype tool which provides a significant foundation for an effective, graphical interface tool for capturing and representing structural knowledge for a wide range of physical systems.

4.5 Contributions to Distributed Artificial Intelligence

The third area in which this report presents results is in the advancement of distributed artificial intelligence. A significant portion of our research effort has concentrated on issues in the design of distributed, multi-agent systems. We have focused on investigating cooperation paradigms and have implemented three distinct cooperative, distributed AI systems.

The Distributed Multi-Agent Planner (DMAP) demonstrates, in the context of the service restoral problem, the capability to design a *truly distributed* planning system. In this system there is no central node, no locus of control, and no complete, global knowledge. The agents exchange messages in order to coordinate actions and gain sufficient knowledge to make coherent, globally satisfactory decisions. We present experimental results which show that not only is such a design feasible, but also that the message exchange does not produce the equivalent of complete, global knowledge.

We have studied cooperation through the sharing of inferences in a common, local knowledge base. This work resulted in the implementation of a truth maintenance system (the MATMS) which enables multiple agents to maintain different, possibly inconsistent, beliefs while insisting on a single, shared, logically consistent knowledge base.

The Distributed Automated Reasoning System (DARES) was implemented to investigate cooperative strategies for exchange of knowledge among agents working toward a single goal, but with each agent having only partial knowledge locally. DARES may be viewed as a prototypical system at this stage, but it is intended to serve as a model for such problem solving activities as distributed situation assessment which must be performed by the performance assessment agents.

4.5.1 A Distributed Multi-Agent Planner (DMAP)

In this section of the report, we present our work on a distributed multi-agent planning system, DMAP. Distributed planning can be described as an activity in which a group of semi-autonomous agents, each of which has a limited view of the global system state and control over only a subset of the resources required to execute an acceptable plan, cooperatively arrives at a set of actions that satisfy some goal or group of goals. We view distributed planning as a process that proceeds in two phases: plan generation and negotiation. We describe each of these phases in the sections which follow.

An important component of this planning activity is distributed plan generation; we have developed a mechanism for performing this task in a class of problems in which resource allocation can be viewed as a planning problem. In distributed environments such as these, problem solving agents must cooperate to incrementally build plans to complete tasks without knowing *a priori* what resources are needed or how they can be utilized. Therefore, distributed plan generation involves properly assessing which local resource allocations are associated with a single global plan and which are parts of distinct global plans. Our solution to the problem requires that an agent only be aware of a limited and abstracted view of global plans.

In many domains, planning can be viewed as a form of a distributed resource allocation problem, in which actions make use of resources that are objects available for use in satisfying system goals. The resources available generally have three significant characteristics: resources are indivisible (not consisting of component resources), the supply of resources is limited, and use of these resources cannot be time shared for concurrent satisfaction of multiple goals.

Our model of planning differs from many others in that both control over resources and knowledge about these resources are distributed among problem solving agents. Some of the resources are under the direct control of a single agent, while control over others is *shared* by two agents. Resources controlled by a single agent are *local* to that agent and cannot be allocated by any other agent. Indeed, any given agent only has knowledge concerning those resources that are local and those whose control it shares. Shared resources are also local to an agent but must be regarded somewhat differently in reasoning because allocation of shared resources must be coordinated by those agents that share control. Each agent must therefore know which of its local resources are shared and which agents are involved in the shared control of a particular resource.

In this kind of environment, global goals may arise concurrently in multiple agents. The system objective is one of finding a set of resource allocations that satisfies as many of the global goals as possible, subject to constraints.

In the sections which follow, we first discuss distributed plan generation in more detail and describe one solution to the problems that arise with the aid of an example.

These problems arise because one agent may be asked to extend the same partial plan several times. Thus an agent must be aware of situations in which it is extending a partial plan it has already participated in constructing. Since no agent ought to have knowledge of an *entire* global plan, the central issue is one of determining a mechanism whereby an agent can acquire this type of awareness *without* requiring that it have detailed knowledge about global plans.

We then describe a formalism that has been developed for abstracting and propagating information about nonlocal impact of decisions made locally. Our work provides mechanisms for determining impact at three levels: locally on the level of plan fragments, locally on the level of goals, and nonlocally.

4.5.1.1 Distributed Plan Generation We view the objective of distributed plan generation as one of determining sequences of local actions that can be performed in a coordinated fashion by distributed agents to satisfy several global goals. Thus, the collection of local actions that satisfies a *single* global goal constitutes a global plan that exists as plan fragments distributed among the agents. A plan fragment, then, is a sequence of operator applications to objects under the control of an agent that would transform the global system, possibly through intermediate states, to a new state. An agent can extend a plan fragment if the agent can create a plan fragment which would transform the system from the new state to a state that is closer to the goal state. The knowledge concerning what state transformations each agent can make is distributed among the agents thereby making it impossible, in most cases, for a single agent to devise a global plan.

Plan generation begins when an agent is notified of the instantiation of a global goal. The agent creates a subgoal corresponding to this global goal and determines all sequences of actions it can take to bring the system to a state that locally appears closer to the goal state. Each of these alternatives becomes a *plan fragment.* If any of these plan fragments would modify the system state to a new state that is not the goal state, the agent must issue requests for extension of the plan by agents that may be able to transform the system from the new state to the goal state or a state that may be nearer to the goal state. This process is repeated until all requests are processed.

It is clear that every request to extend a plan must carry certain information which will permit an agent to achieve a new state which locally appears closer to the goal state. Specifically, a request must contain the identification of the associated global goal, a description of that goal, and a description of the new state.

It is essential to observe that during plan generation, a given agent may be asked to add a new sequence of actions to the same global plan several times. For example, Agent A may formulate a plan fragment that would transform the system to state 1, then request that Agent B extend the plan. Agent B may devise a sequence of

actions that would bring the system from state 1 to state 2 and then request that Agent C extend the plan, whereupon Agent C may create a plan fragment that would transform the system to some new state. Agent C could then request that Agent A extend the plan. Clearly, it is necessary for an agent to be able to detect when it is being asked to build another piece of a global plan it has already partially constructed. If the agent has already built one or more parts of a plan, it must know which plan fragments were previously used. This information is needed for two reasons. First, the agent must not inadvertantly create a plan which would bring the system to the same state twice. Secondly, the agent must be able to accurately assess which action subsequences belong to the same plan fragment and which belong to totally different plan fragments, so that interactions among plan fragments can properly be determined.

One mechanism for providing an agent with the means for acquiring this awareness involves attaching a list of support names to each plan fragment. The support names represent abstractions of the global plans associated with a plan fragment. They are incrementally constructed, with each agent appending a "tag" to identify its own plan fragments. The tags allow each agent to reason about its utilization of a particular plan fragment in a global plan. They *do not* embody information which allows an agent to reason about the participation of other agents in a particular global plan. The discipline used in constructing support names guarantees that if, at the end of plan generation, a given plan fragment has no associated support names then no acceptable global plan uses this plan fragment.

Support names follow a plan as it is developed by the agents. Once a plan has been completed, the requisite plan fragments can be marked by tracing continuation requests using the support name. Thus, an agent can determine which requests are part of the same global plan and which belong to distinct global plans. If it is determined that a plan cannot be completed, the appropriate support names can be deleted.

4.5.1.2 An Example

As an example, consider the communications network shown in Figure 12. There are five problem solving agents each controlling part of a network of geographically distributed communication links. The circles represent communication sites and the lines joining sites represent communication links. In this domain, the problem of restoring disrupted service can be viewed as a planning problem in which one operator, Allocate, is utilized to allocate communications resources. The resources are links and a global plan is a sequence of local link connections which restores communication between two sites. Thus, a partial plan or plan fragment involves an allocation of resources that transforms the system from a state in which it has a path ending at one site to one in which it has a new path ending at another site.

Table 1 summarizes the knowledge about resources available to each agent and the associated control relationships.

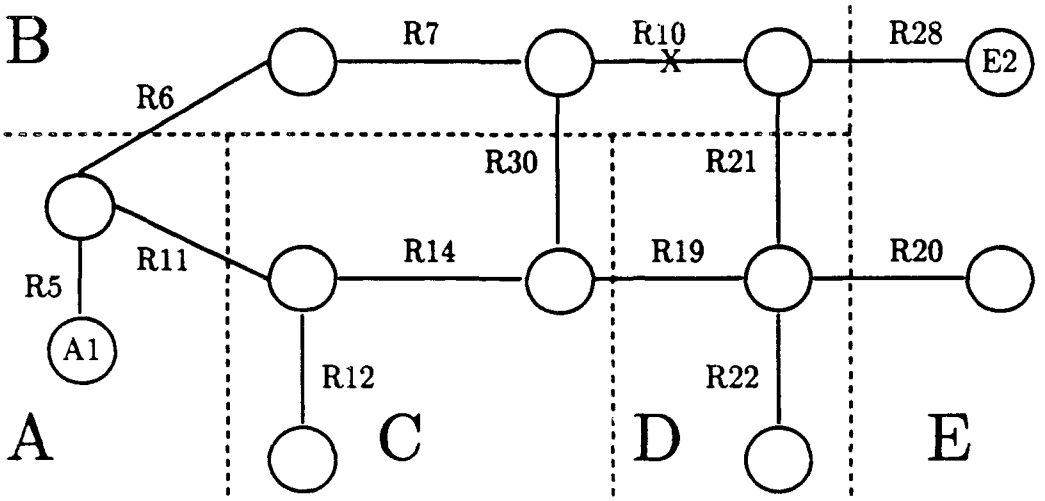


Figure 12: Example Network

Agent	Resources
A	R5 R6 R11
B	R6 R7 R10 R21 R28 R30
C	R11 R12 R19 R30
D	R10 R19 R20 R21
E	R20 R28

Table 1: Local Resource Control

Let us assume that originally communication between sites E2 and A1 follows a path over links R28-R10-R7-R6-R5 but that at some point link R10 fails and communication between sites E2 and A1 must be restored over a different route. Furthermore, let us assume that Agent E is notified that a global goal to restore the path between sites E2 and A1 has been instantiated.

It is perhaps easiest to explain how plan generation proceeds by viewing the activities of agents at global time slices.

T1:

- Agent E creates a subgoal to restore a path from site E2 to site A1 and determines that it has one plan fragment that locally satisfies this subgoal:

pfE1 that uses R28. Since this plan fragment does not satisfy the global goal, Agent B is requested to find a path to site A1 using R28 with support name (E1).

T2:

- Agent B: Creates a subgoal using pfB1 using R28-R21 with support name (E1)

  Requests: Agent D use R21 with support name (B1 E1)

T3:

- Agent D: Creates a subgoal using pfD1 using R21-R20 with support name (B1 E1) and pfD2 using R21-R19 with support name (B1 E1)

  Requests: Agent E use R20 with support name (D1 B1 E1) Agent C use R19 with support name (D2 B1 E1)

T4:

- Agent C: Creates a subgoal using pfC1 using R19-R11 with support name (D2 B1 E1) and pfC2 using R19-R30 with support name (D2 B1 E1)

  Requests: Agent A use R11 with support name (C1 D2 B1 E1) Agent B use R30 with support name (C2 D2 B1 E1)

- Agent E: Determines it has no way to extend Agent D's request

  Notices: Agent D remove support name (B1 E1) from pfD1

T5:

- Agent A: Creates a subgoal using pfA1 using R11-R5 with support name (C1 D2 B1 E1). This plan fragment completes the global plan.

  Notices: Agent C mark support name (D2 B1 E1) for pfC1 acceptable

- Agent B: Creates a subgoal using pfB2 using R30-R7-R6 with support name (C2 D2 B1 E1)

  Requests: Agent A use R6 with support name (B2 C2 D2 B1 E1)

- Agent D: Removes support name (B1 E1) from pfD1 however since pfD2 uses this support name, no propagation occurs.

T6:

- Agent A: Creates a subgoal using pfA2 using R6-R5 with support name (B2 C2 D2 B1 E1). This plan fragment completes the global plan.

  Notices: Agent B mark support name (C2 D2 B1 E1) for pfB2 acceptable

- Agent C: Marks support name (D2 B1 E1) for pfC1 acceptable
  Notices: Agent D mark support name (B1 E1) for pfD2 acceptable

T7:

- Agent B: Creates a new plan fragment, pfB3 using R6-R7-R30 and R21-R28 with support name (C2 D2 B3 E1). Removes support name (C2 D2 B1 E1) from pfB2.
  Notices: Agent C add support name (D2 B3 E1) to pfC2 Agent C remove support name (D2 B1 E1) from pfC2
- Agent D: Marks support name (B1 E1) for pfD2 acceptable
  Notices: Agent B mark support name (E1) for pfB1 acceptable

T8:

- Agent B: Marks support name (E1) for pfB1 acceptable
  Notices: Agent E mark pfE1 acceptable
- Agent C: Adds support name (D2 B3 E1) to pfC2. Removes support name (D2 B1 E1) from pfC2 however, pfC1 uses this support name so no propagation takes place.
  Notices: Agent D add support name (B3 E1) to pfD2

T9:

- Agent D: Adds support name (B3 E1) to pfD2
  Notices: Agent B add support name (E1) to pfB3
- Agent E: Marks pfE1 acceptable

T10:

- Agent B: Adds support name (B3 E1) to pfB3

Table 2 shows the plan fragments created by each agent, the resources used by these plan fragments, and the support names associated with each plan fragment at the end of plan generation. The "*" in the support names column for pfE1 denotes that this plan fragment has been marked acceptable. Note that pfD1 has no support names because this plan fragment is not part of any acceptable global plan. Agent B has determined that no acceptable global plan uses pfB2 alone; rather, if this plan fragment is used, it is always used in conjunction with pfB1. Therefore, Agent B has created a new plan fragment, pfB3, which uses the resources of both pfB1 and pfB2 and Agent B has given this new plan fragment support. Since no acceptable global plan uses pfB2 its support names are removed. PfB1 on the other hand, is part of a global plan that does not use other plan fragments in Agent B so the support names for pfB1 are not removed.

Agent	Plan Fragments	Resources	Support Names
A	pfA1	R11–R5	(C1 D2 B1 E1)
	pfA2	R6–R5	(B2 C2 D2 B1 E1)
B	pfB1	R28–R21	(E1)
	pfB2	R30–R7–R6	none
	pfB3	R28–R21 R30–R7–R6	(C2 D2 B3 E1)(E1)
C	pfC1	R19–R11	(D2 B1 E1)
	pfC2	R19–R30	(D2 B3 E1)
D	pfD1	R21–R20	none
	pfD2	R21–R19	(B1 E1)(B3 E1)
E	pfE1	R28	*

Table 2: Plan Generation Results

4.5.1.3 Experimental Results Research in distributed planning is currently being conducted in the context of the communications domain described in the previous example. The implementation model, however, contains much more of the detail associated with a real world communications network [10]. Local searches for plan fragments are not simple searches for paths of links in and out of a subregion as might be assumed given the example above. On the contrary, local searches involve tracing through complex interconnections of various types of communications equipment at the sites within a subregion. The knowledge base that is searched contains information about physical communications equipment such as radios, various levels of multiplexers, and computer controlled switching devices called DPAS's. This equipment is connected in various configurations at each site within each subregion. Paths through the subregion must be traced through various levels of multiplexing and demultiplexing as they pass through long chains of communication equipment.

Existing planners use several different architectures and, moreover, the level of abstraction at which planning occurs varies from system to system. Experiments have been conducted so that distributed plan generation as performed in DMAP may be compared to plan generation schemes with various architectures using different levels of abstraction. In each of the tested schemes, an agent which has control over part of a network has detailed information about that part of the network and *only* that part of the network. If any other information is used for plan generation, it is either abstract knowledge in the form of plan fragments, or limited, abstract knowledge in the form of support names. The following is a description of the plan generation paradigms used in these experiments. The first is a single agent system and the rest are multiple agent systems.

Single Agent/Detailed Global View (SA/DGV) A single agent is responsible for the entire system rather than distributing system knowledge among multiple agents. In this approach, a local search for plan fragments is equivalent to a global search for global plans that will satisfy system goals. This will be referred to in the text as the Single Agent strategy.

Multiple Agent/Limited Abstract Global View (MA/LAGV) This is the approach described in this paper. Plans are constructed by multiple agents which have an incomplete, limited view of the global plans. This incomplete, limited view is determined by the incremental construction of support names, and therefore is different at each agent in the system. This will be referred to in the text as the Localized strategy.

Multiple Agent/Central Abstract Global View (MA/CAGV) Agents use descriptions of the circuits which are to be restored to determine all the possible ways they might be able to participate in a global plan. Circuit descriptions include the circuit name, priority, source and destination. Each agent is given this information for each circuit which is to be restored. Each agent then returns partial routes through its subregion which could be used in a global path. The results are returned in the form of plan fragments using local and shared resources. The results of these local searches are sent to a single agent who pieces the plan fragments together by their use of shared resources into acceptable global plans. Once this is completed, each agent is notified of its participation in global plans. The view of this single planning agent is not limited in the sense that it does know about the complete set of plan fragments in the system. However, its view is abstract since this agent knows nothing about the details of the communications equipment and interconnections at each site. This will be referred to in the text as the Centralized strategy.

Multiple Agent/Replicated Abstract Global View (MA/RAGV) As in the Centralized approach, local searches are conducted by each agent using high level circuit descriptions. The results of these searches, the local plan fragments, are sent to every other agent in the system. Then, with complete knowledge of every plan fragment in the system, each agent forms the global plans and determines its own role in each. This will be referred to in the text as the Replicated strategy.

The key parameters monitored in these experiments are the simulated time required to generate plans, the average CPU time required by each processing node to generate plans, and the amount of message traffic sent during the simulation.

In addition, three network configurations were chosen to observe the effect of various topological extremes. In this domain, the network topology actually defines

the complexity of the roles of agents in the multiple plan decompositions. Therefore, by varying these topological extremes it is also possible to observe the performance of these strategies when agent participation takes on roles of different complexity. Each network contains twelve sites divided into five subregions with various inter- and intrasubregion connectivity. Figure 13 shows the configuration where the subregions are connected in a straight line and Figure 14 shows the subregion connections which form a ring. The third topology chosen is shown in Figure 15. Here each subregion is connected to every other subregion creating a tightly coupled network.

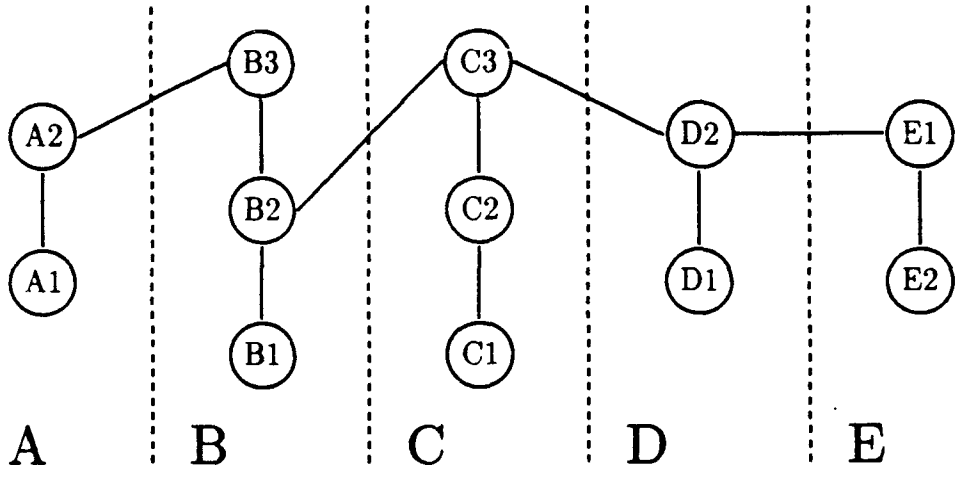


Figure 13: Line Topology

The results of these experiments are shown in Figures 16,17, and 18. As expected, the Single Agent strategy performs the worst in terms of the time taken to devise global plans. This observation holds true over each of the tested topologies. This points to the desirability of distributed multiagent systems over centralized single agent systems when the systems are large.

The Centralized, Replicated, and Localized strategies all take about the same amount of time to determine global plans for the line topology. As well, the CPU time per agent is approximately the same. However, the amount of message traffic required by the Replicated strategy exceeds that of both the Centralized and Localized strategies with the Centralized strategy performing better as the number of goals grows.

For the ring topology, the CPU time per agent for the multiagent strategies begins to separate with the Centralized strategy clearly performing better as the number of goals increases. The Replicated and Localized strategies appear to be following approximately the same line. Regarding the time to construct global plans, the

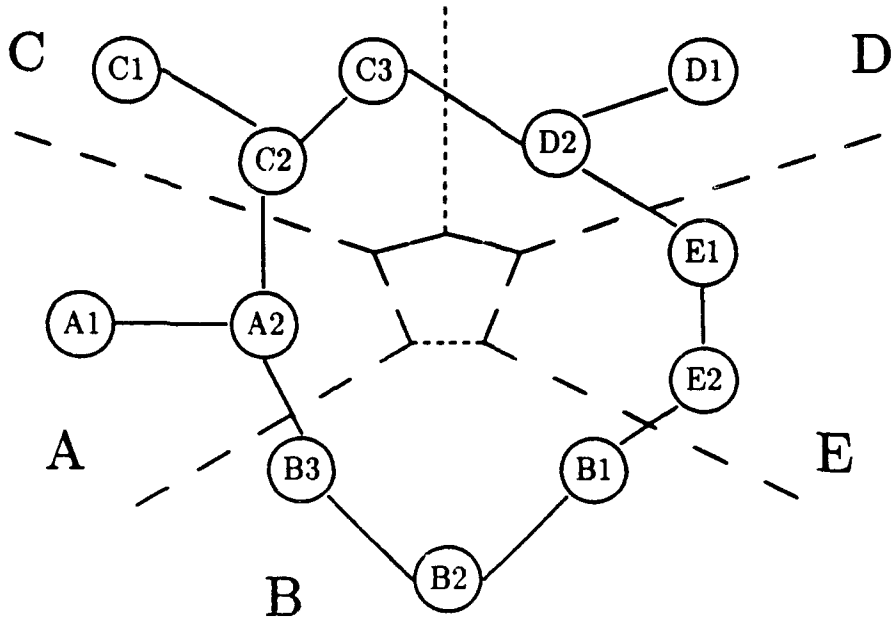


Figure 14: Ring Topology

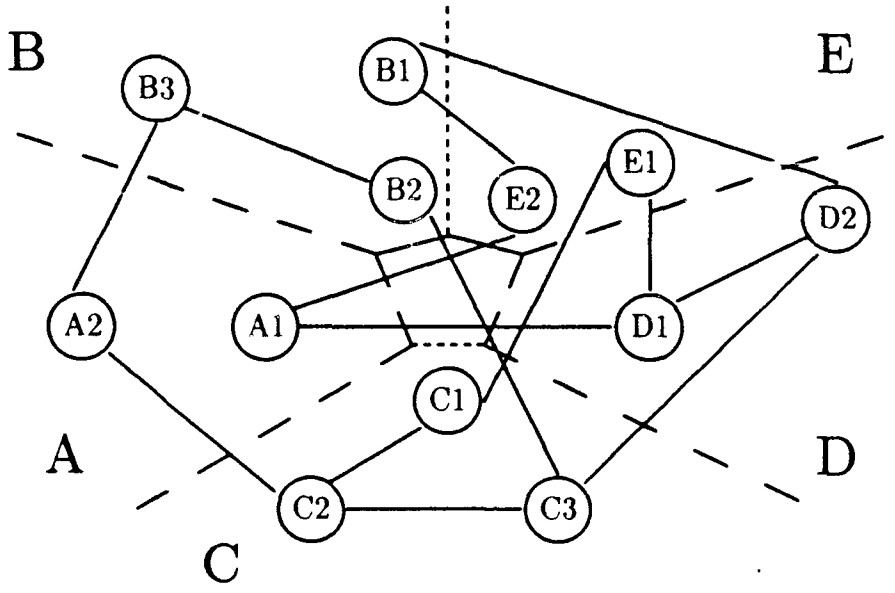


Figure 15: Tightly Coupled Topology

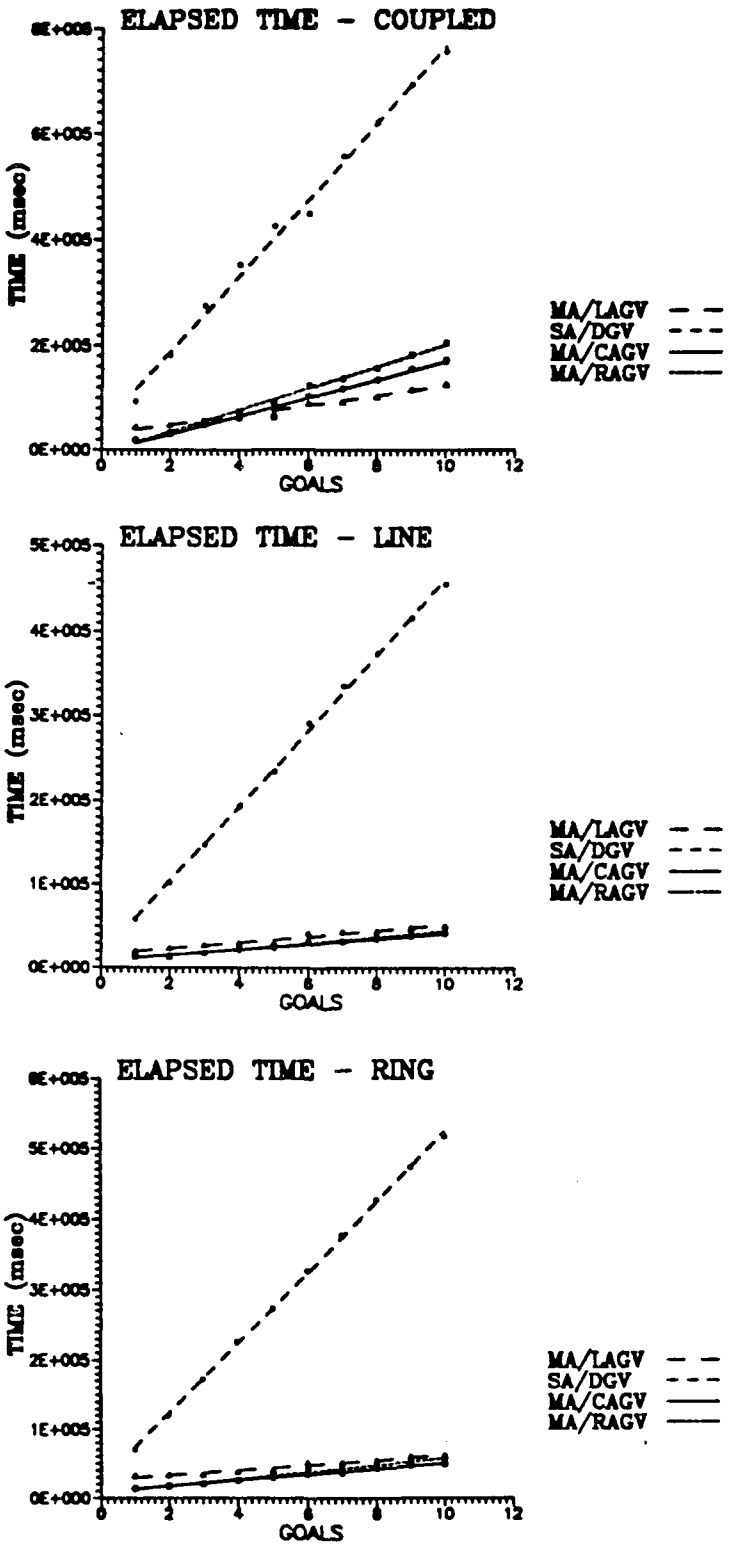


Figure 16: Experimental Results: Simulated Elapsed Time

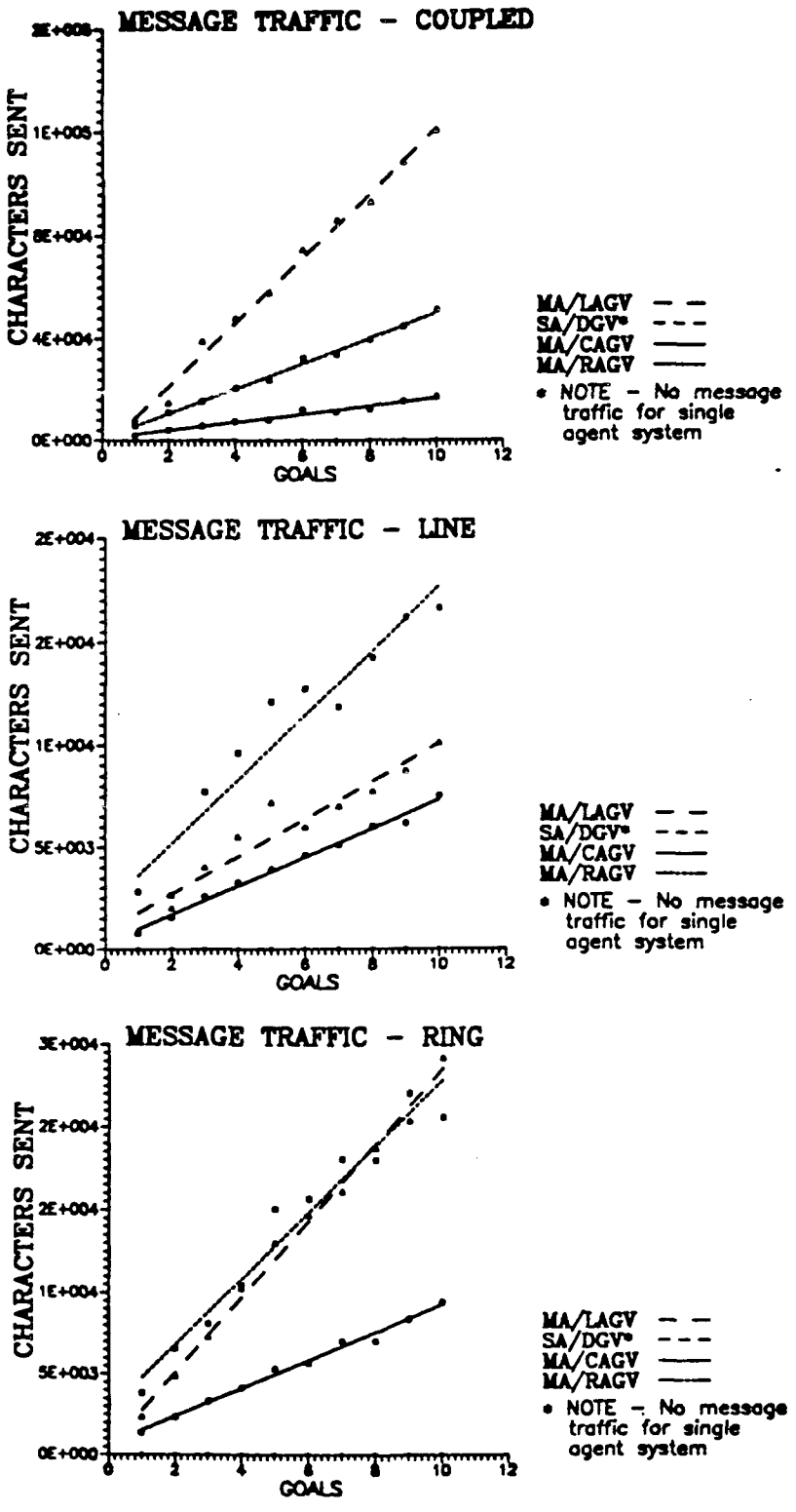


Figure 17: Experimental Results: Message Traffic

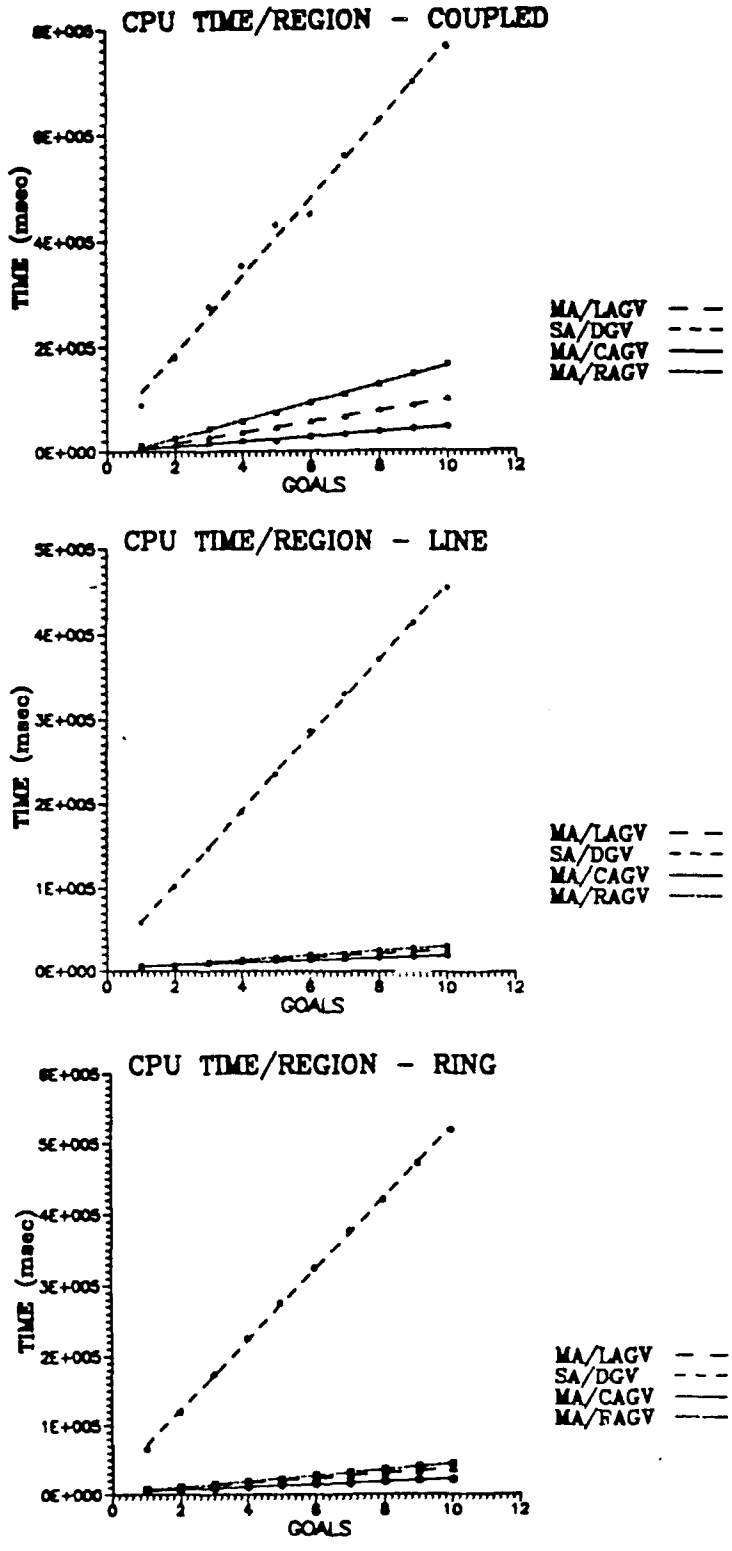


Figure 18: Experimental Results: CPU Time/Region

Replicated and Centralized strategies outperform the Localized when the number of goals is small. However, as the number of goals increases, the lines appear to be converging. The results for the message traffic required shows that the Replicated and Localized strategies have approximately the same requirements while the Centralized strategy requires less message traffic.

When the topology is tightly coupled, the strategies perform with significant differences. The Localized strategy clearly requires less time than both the Centralized and Replicated to devise plans as the number of goals increases. However, the CPU time per agent required is clearly less for the Centralized strategy with the Localized strategy coming in second and the Replicated performing worse. In addition, there is a marked difference in the amount of message traffic required by the different strategies. The Localized strategy requires the most message traffic, the Replicated less, and the Centralized still less.

For the network topologies tested, there is a clear question of trade-offs. For the ring and line topologies, the Centralized strategy performs better overall. The price paid however is vulnerability. In domains where survivability is an important concern, such as a military communications network, the Centralized strategy obviously is undesirable because of the dependence upon a single agent. If for some reason this single agent fails, mechanisms would necessarily have to be built in to detect the failure. Furthermore, all the work which had been performed by this agent would have to be recalculated. For the tightly coupled topology, the Localized strategy will take less time to construct plans but the price paid is in the amount of message traffic required.

4.5.1.3.1 Performance Analysis The performance of distributed plan generation can be analyzed by considering the time required to generate plans and the amount of message traffic sent.

The time required to generate plans is influenced by factors on two levels. At one level, this parameter is dependent upon the amount of time required to pass the plan among each of the agents involved in its construction. Therefore, from a global viewpoint, the time required to generate plans is directly related to the length of the longest chain of agents involved in building a plan. At another level, the amount of time required to generate plans is determined by the processing time of each individual agent. As the relations between requests to extend a plan and multiple plan decompositions become more complex, so does the processing involved to determine distinct alternatives. Thus, from a global perspective, the time required to generate plans is also directly related to the complexity of the roles of agents in multiple plan decompositions.

The message traffic necessary for plan generation is also directly related to the participation of agents in multiple plan decompositions. When an agent is notified that a

plan it has helped to build has been deemed acceptable, that agent is responsible for the propagation of this information. If the agent participated only once in the plan construction, a single message is required to continue the propagation. However, if the agent participated multiple times in the construction, then two messages are sent, one to propagate the new support name and one to remove the old support name. Thus, the message traffic required to generate plans increases as the complexity of the roles of agents in multiple plan decompositions increases. However, it should be noted that the amount of message traffic required does not approach that which would be needed to transmit complete, detailed global information to each agent in the system.

These experiments illustrate that distributed plan generation can be accomplished by passing merely a limited amount of information among system agents. The only information required includes descriptions of the goal state and the present state of the plan, and information which allows agents to determine their previous actions in the construction of the plan. This last piece of information is provided by the implementation of support names. Experimentation shows that building a complete, detailed global view at any agent is unnecessary. Plan generation using support names will perform best in domains where goals can be satisfied through the actions of a small number of agents who participate in the construction of plans only once. The performance of distributed plan generation using support names will degrade as the number of requests for extensions of plans grows and as the number of times agents participate in building the same plan increases.

4.5.1.4 Multistage Negotiation As described earlier, we view distributed planning as consisting of two phases: plan generation and negotiation. For each global goal, plan generation determines a set of plans each of which is feasible, taken in isolation. Each of these plans consists of a collection of *plan fragments*, one resident in each participating agent. Plan fragments which collectively form a (global) plan must embody a consistent choice of resource allocations. This consistency is reflected in the presence of constraints that enforce a coordinated allocation of resources that are shared among agents. It is important to note that when a resource r is shared by two agents (agent A and agent B), allocation of r for use in satisfying a goal must occur in *both* agent A and agent B.

To motivate this assumption, consider a simple example in which there is a work surface shared by two robots, each of which has a private collection of blocks. Suppose that blocks come in various colors, shapes, and sizes. A goal in the context of this example is to construct a stack of four blocks all of which have some common property subject to the constraint that no block can be placed on a block that is smaller than itself.

In the context of this example, the blocks can be modeled as resources, and since each robot has its own private collection of blocks, neither robot knows the number,

size, shape, or color of blocks the other robot controls. To avoid a collision of the two robots, the time that a robot can be in the work space is modeled as a shared resource. A plan fragment consists of a sequence of one or more placement operations performed by a single robot at specific times using blocks from its own private supply. A global plan consists of a sequence of four placement operations that construct a tower satisfying the goal description. Depending on the distribution of blocks in the robots' private supplies, a global plan will consist of placement operations performed by the two robot arms in various combinations.

When we assume that there may be several goals the system is attempting to satisfy concurrently (e.g. several towers to be built), it becomes clear that it may not be possible to satisfy *all* of the global goals because of resource allocation conflicts. Negotiation is necessary to select a set of plans that satisfies as many global goals as possible.

Multistage negotiation [7] has been developed as a means by which an agent can acquire enough knowledge to reason about the impact of local activity on nonlocal state and modify its behavior accordingly. This protocol can be viewed as a generalization of the contract net protocol [43, 44, 13]. It produces a cooperation strategy that is similar in character to the Functionally Accurate, Cooperative (FA/C) paradigm [32], in which nodes iteratively exchange tentative and high level partial results of their local subtasks.

In the sections which follow, we describe a formalism that has been developed for abstracting and propagating information about nonlocal impact of decisions made locally. Our work provides mechanisms for determining impact at three levels: locally on the level of plan fragments, locally on the level of goals, and nonlocally. We first present formal definitions that characterize this impact. We then give algorithms for the construction of sets that reflect various levels of impact and analyze their complexity in cases for which the problem is likely to be overconstrained and those for which it is likely to be underconstrained. Finally, we reflect on the level of transaction activity required to propogate local information nonlocally.

4.5.1.5 Reasoning About Constraints and Conflicts For the purposes of illustrating our definitions, we consider a scenario involving four agents in a distributed system cooperatively attempting concurrent satisfaction of four goals. A number of global plans have been constructed during plan generation, as indicated in Table 3. In Table 3, each goal is identified by g*i* ($i = 1, 2, 3, 4$). The set of alternative plans for each specific goal g*k* are identified by g*k*p*l* ($l = 1, 2, \ldots$). Thus we see that goal g1 has five distinct alternative plans, g1p1, g1p2, etc.

It should be noted that Table 3 shows the global plans from a *global* perspective. No single agent in a distributed problem solving system has complete knowledge concerning any of these plans. Indeed, unless some system goal can be satisfied by a

plan	plan fragments	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11
g1p1	A-a B-a C-a D-a	1	1			1	1			1		
g1p2	A-a C-a D-b	1	1			1		1	1			
g1p3	A-c C-c	1	1		1							
g1p4	A-b B-b C-b	1			1						1	1
g1p5	D-c						1		1			
g2p1	A-d C-d	1	1	1	1							
g2p2	A-e B-c D-d	1						1		1		1
g2p3	C-e D-e			1	1	1	1		1			
g3p1	A-f C-g D-f	1	1	1		1		1				
g3p2	A-f C-g D-g	1	1	1		1	1					
g3p3	C-g D-f				1	1		1				
g3p4	C-h D-g				1	1	1					
g4p1	A-g B-d	1										1
g4p2	A-g B-e C-j	1		1	1						1	1
g4p3	A-h C-i D-h		1			1						
g4p4	A-h C-i D-i		1			1						
g4p5	C-k D-h				1	1		1				
g4p6	C-k D-i				1	1	1					

Table 3: Global Plans Generated

single agent using its own local resources, no single agent is even aware of the total number of alternative plans that have been generated.

From Table 3, it is evident that global plans are composed of collections of local *plan fragments*. For instance, global plan g3p3 is composed of plan fragments C-g and D-f. Plan fragment C-g denotes a set of local actions that agent C could take in partial satisfaction of goal g3. Satisfaction of g3 using g3p3 would require the actions D-f by agent D as well as the set of actions C-g in agent C.

Local knowledge about plan fragments is shown in Table 4. Notice that if the entry on the *resource count* line for resource r in agent i is k, then agent i has k copies of resource r to utilize in problem solving. The shared resources are evident, as they are known to more than one agent. Observe that *r10* is a shared resource. There is only one copy of *r10* in the system, and its allocation must be jointly controlled by agents B and C.

It is important to note that each agent has *only* the local knowledge about plan fragments shown in Table 4. This means, for example, that agent A is aware that

Agent A				
goal	plan frag	r1	r2	r11
resource count		3	2	2
g1	A-a	1	1	
	A-b	1		1
g2	A-d	1	1	
	A-e	1		1
g3	A-f	1	1	
g4	A-g	1		1
	A-h		1	

Agent C						
goal	plan frag	r2	r3	r4	r5	r10
resource count		2	3	2	2	1
g1	C-a	1			1	
	C-b			1		1
	C-c	1		1		
g2	C-d	1	1	1		
	C-e		1	1	1	
g3	C-g	1	1		1	
	C-h			1	1	
g4	C-i	1			1	
	C-j		1	1		1
	C-k			1	1	

Agent B				
goal	plan frag	r9	r10	r11
resource count		1	1	2
g1	B-a	1		
	B-b		1	1
g2	B-c	1		1
g4	B-d			1
	B-e		1	1

Agent D						
goal	plan frag	r5	r6	r7	r8	r9
resource count		2	2	1	3	1
g1	D-a	1	1			1
	D-b	1		1	1	
	D-c		1		1	
g2	D-d			1		1
	D-e	1	1		1	
g3	D-f	1		1		
	D-g	1	1			
g4	D-h	1		1		
	D-i	1	1			

Table 4: Local Knowledge About Plan Fragments

plan fragment A-b for goal g1 coordinates with s ne plan fragment known to agent B as a component in some global plan or plans in satisfaction of g1. Agent A knows this because resource r11 is shared between agents A and B. Agent A *does not know* anything about plan fragments that are local to agent B.

To enable an agent to efficiently exchange knowledge concerning the nonlocal impact of local decisions, we determine a *conflict set* for each plan fragment. We then use the conflict set to construct an *exclusion set* for each plan fragment that reflects the potential impact on an agent's ability to participate in satisfying other goals, assuming that plan fragment x is executed. At the highest level of abstraction, we use exclusion sets to form *infeasibility sets.* Knowledge summarized in its infeasibility sets allows an agent to reason about the way in which its decision to satisfy one goal may affect its ability to satisfy other goals. Finally, we propagate these local concepts to other agents with the construction of *induced exclusion sets.*

Before formalizing these concepts, we describe our notational conventions in the next section.

4.5.1.5.1 Notation

- We define maximal and minimal subsets of sets whose elements are sets in the standard way. Given a set of sets $S = \{S_1, \ldots, S_n\}$ with a partial order $<$ defined on subsets of S in the standard way (that is, $S_i < S_j \Leftrightarrow S_i \subseteq S_j$), we say that S_i is **maximal** if $\nexists S_j \ni: S_i < S_j$. Furthermore, S_i is **minimal** if $\nexists S_j \ni: S_j < S_i$.
- $P_A = \{$ all plan fragments known to agent A $\}$.
- If $pf_x \in P_A$, then pf_x is associated with satisfaction of some goal $g(pf_x)$.
- The set of goals known to agent A is $G_A = \{g \mid g = g(pf_x)$ for some plan fragment $pf_x \in P_A\}$.
- For each goal g in G_A, there is an associated set of plan fragments $pfs_g = \{x \mid x \in P_A$ and $g = g(x)\}$.
- $copies(r_i)$ denotes the number of copies of resource r_i available for use by agent A.
- $resources(pf_x)$ denotes the resources required to execute plan fragment pf_x.
- $r_i(pf_x)$ denotes the number of copies of resource r_i needed by plan fragment pf_x.
- A set of plan fragments in Agent A, $P = \{pf_1, \ldots, pf_n\}$ is said to be **compatible** if $\sum_{k=1}^{n} r_i(pf_k) \leq copies(r_i)$ for all i and $g(pf_j) \neq g(pf_k)$ for $j \neq k$.

- **A maximal compatible set of plan fragments in A relative to pf_x** is a maximal subset of $S_x = \{P \mid P$ is a compatible set of plan fragments and $pf_x \in P\}$.

4.5.1.5.2 Formal Definitions The conflict set for plan fragment pf_x indicates the minimal impact (locally) of a choice to execute pf_x. The conflict set for pf_x can be constructed by considering each maximal set M of mutually feasible plan fragments (*including* pf_x) known to an agent. For each such set, M, the complement of M is an element of the conflict set for pf_x.

More formally, the **Conflict Set** for plan fragment pf_x is constructed as follows: Let $X = (P_A - pfs_g) \cup \{pf_x\}$, where $g = g(pf_x)$. For each maximal compatible subset M of plan fragments in A relative to pf_x, the set $X - M$ is a member of the conflict set for pf_x. Thus, $CS_{pf_x} = \{c \mid c = X - M$, where M is a maximal compatible subset of plan fragments in A relative to pf_x $\}$.

To illustrate this formalism, we compute the conflict set for D-b in our example scenario. The maximal compatible subsets of plan fragments in D relative to D-b are: {D-b, D-e}, {D-b, D-g}, and {D-b, D-i}. Thus the conflict set for D-b is:

{{D-d, D-f, D-g, D-h, D-i}, {D-d, D-e, D-f, D-h, D-i}, {D-d, D-e, D-f, D-g, D-h}}

We are concerned with the conflict set because the conflict set for a plan fragment gives information as to the negative impact of executing that plan fragment. The maximal compatible subsets, on the other hand, indicate maximal sets of feasible choices that are available. There is no reason to believe that an agent should choose some one of these maximal subsets for execution. Indeed, a given agent might never participate in system satisfaction of some of the global goals. (This can be seen in the example scenario by observing that all four global goals can be satisfied through choice of g1p3, g2p3, g3p1, and g4p1. Agent D is only involved through partial satisfaction of g2 and g3.)

Though the view of the conflict set as being formed using the complements of maximal feasible sets is intuitively appealing, when the problem is underconstrained it is computationally more attractive to treat conflict relative to pf_x in a dual form: as the collection of minimal mutually infeasible sets of plan fragments, given that plan fragment pf_x is to be executed.

Three significant observations can be made concerning the conflict set of plan fragment pf_x. First, the complement of each element of the conflict set is indeed a maximal feasible set. Secondly, the agent will be *compelled* to forego execution of all plan fragments in *some* element of the conflict set if it chooses to execute plan fragment pf_x. The local impact of a decision can thus be related to the size of

elements in the conflict set. Finally, representation of impact in the form of a conflict set seems to provide a substantially more compact form of representation that can be more efficiently used in reasoning than many others.

The conflict set for a plan fragment reflects the impact of executing that plan fragment *at the level of mutually infeasible sets of plan fragments.* It is often necessary to reason about the impact that executing a particular plan fragment would have on the potential satisfaction of other goals.

The **Exclusion Set** for a plan fragment, pf_x, is a collection of sets of goals, one of which must be abandoned if pf_x is selected for execution. Thus, if the agent selects plan fragment pf_x then one of the elements of the exclusion set is a set of goals that *cannot* be satisfied through action on the part of this agent. The exclusion set is defined as follows:

For each $s \in CS_{pf_x}$, we define $g_s = \{g \mid pfs(g) \subseteq s\}$. Thus g_s, for an element s of the conflict set, is the set of goals that that cannot be satisfied locally if plan fragments in s are eliminated from consideration. We let $G = \{g_s \mid s \in CS_{pf_x}\}$ and define the **exclusion set for plan fragment pf$_x$**, ES_{pfx}, as the collection of the minimal subsets of G.

Returning to our example, we compute the exclusion set for D-b. The conflict set for D-b has three elements. Using the definition of g_s, we see that

- $g_{D-d,D-f,D-g,D-h,D-i} = \{g3, g4\}$
- $g_{D-d,D-e,D-f,D-h,D-i} = \{g2, g4\}$
- $g_{D-d,D-e,D-f,D-g,D-h} = \{g2, g3\}$

Thus, $G = ES_{D-b} = \{\{g3, g4\}, \{g2, g4\}\{g2, g3\}\}$.

A choice by agent D to execute plan fragment D-a compels agent B to forego local action in partial satisfaction of two of the other three global goals about which it has local knowledge. *Which* two of the three should be abandoned is dependent on decisions made elsewhere.

The exclusion set exposes relationships between plan fragments and goals. It is often desirable to detect and reason about mutually infeasible goals. The relationship of infeasibility is a very strong one. Goal g1 is (locally) infeasible with goal g2 if each of the (local) plan fragments for g1 has g2 in every element of its exclusion set (and conversely). When two goals are (locally) mutually infeasible, an agent knows that it cannot act to satisfy both goals, due to local constraints. Once exclusion sets have been determined, infeasibility is not difficult to detect.

The three types of relationships we have discussed are all rooted in local constraints. Conflict, exclusion, and infeasibility are essentially concepts which would

not be particularly significant were it not for the constraints on joint execution of plan fragments that exist locally. Although the concept of conflict does not appear to propagate in a meaningful manner, exclusion does. The key element in this propagation lies in the observation (which we have made before) that a choice on the part of one agent to satisfy a goal through execution of a specific plan fragment constrains the set of remaining choices that are available to other agents for satisfaction of that goal.

As we have seen, the construction of exclusion sets allows us to assess the impact of executing of a plan fragment that is due to local conflict. In addition, we would like to know how the conflict associated (locally) with execution of a plan fragment affects the ability of other agents to satisfy their goals. The **Induced Exclusion Set** is our mechanism that provides a vehicle for propagating this information by capturing the essence of the impact that local decisions have nonlocally.

In the discussion which follows, we assume that in a distributed environment one agent does not have knowledge concerning another agent's internal state. It specifically does not have any knowledge about resources over which it has no control. The agent must therefore gain knowledge about the impact its choice has on other agents *from those agents*, directly or indirectly.

The **Induced Exclusion Set** for a plan fragment, pf_x, in Agent A, is a collection of sets of goals, one of which must be abandoned by one or more non-local agents if Agent A executes pf_x. The induced exclusion set for pf_x, IE_{pfx}, is defined in the paragraphs which follow.

Let $X_{pfx} = \{pfi \mid pfx \in P_A,\ pfi \notin P_A,\ resources(pfx) \cap resources(pfi) \neq \phi,$ and $g(pfx) = g(pfi)\ \}$. Thus, each individual plan fragment in X_{pfx} is a non-local plan fragment which may connect directly with pfx (via a shared resource) in some global plan.

For each agent, K, with plan fragments in X_{pfx} we must determine the contribution to the induced exclusion set for pfx due to constraints known by agent K. For each plan fragment $p \in X_{pfx} \cap P_K$, we therefore let

$$e_p = \{e \mid e = es \cup ie \text{ for } es \in ES_p \text{ and } ie \in IE_p\}$$

Notice that each e_p is a set of sets, each of whose members reflects potential conflicts that could arise if plan fragment p is selected by agent K. In this construction, each es represents a contribution to e_p that reflects constraints local to agent K, while each ie denotes a contribution that agent K has received from other agents relative to plan fragment p. For this reason, it is necessary to combine these contributions into a single element, $E_{K,pfx}$, that may be propagated to Agent A. $E_{K,pfx}$ is defined as the collection of minimal subsets of $\bigcup e_p$ for $p \in X_{pfx} \cap P_K$.

Continuing the definition, the **induced exclusion set for plan fragment pfx**, IE_{pfx} is the collection of the maximal subsets of $E = \bigcup E_{K,pfx}$. This definition of IE_{pfx} permits incremental construction of induced exclusion sets under the assumption that initially $IE_{pfx} = \phi$ for all plan fragments.

Once again, returning to our example, we compute the induced exclusion set for C-a. Observe that plan fragment C-a matches (in D) with either D-a or D-b and in A with A-a, so that X_{C-a} = {A-a,D-a,D-b}. As we have seen, the exclusion set for D-b is {{g3, g4}, {g2, g4}, {g2, g3}}. Coincidentally, the exclusion set for D-a is the same as that for D-b, while for A-a the exclusion set is {{g2}, {g3} ,{g4}}. The set E used in computing the induced exclusion set for C-a is the union of the exclusion sets just mentioned, so $E = \{\{g2\}, \{g3\}, \{g4\}, \{g3, g4\}, \{g2, g4\}, \{g2, g3\}\}$. The induced exclusion set for C-a is the set of maximal subsets of E, so $IE_{C-a} = \{\{g3, g4\}, \{g2, g4\}, \{g2, g3\}\}$.

Intuitively, this is telling agent C that agent A is forced to forego one other goal if C-a is chosen and agent D is forced to forego two of the other three goals if C-a is selected. Each nonlocal agent transmits a minimal set of exclusions it knows about. Clearly, agent D reports more extensive nonlocal impact, and the construction of the induced exclusion set via maximal subsets reflects this impact.

The induced exclusion set is incrementally built during negotiation. When one agent (agent A) requests information about the impact of executing plan fragment pf_x on another agent (agent B), agent B attempts to summarize all the knowledge it has about that impact. This knowledge is initially found in the exclusion sets of each of its plan fragments which coordinate with plan fragment pf_x. As has been mentioned, the induced exclusion set in agent A for plan fragment pf_x is empty initially. As nonlocal knowledge becomes available, this set is augmented in the obvious way. Given sufficient time, an agent can acquire knowledge about the system wide impact of executing each of its plan fragments. It does so, however, *without* the exchange of detailed information concerning resource availability in the system. It is not difficult to show that incremental construction of the induced exclusion set for a plan fragment can be managed so that it converges after no more than $2n$ exchanges of information, where n is the number of agents in the system.

4.5.1.6 Computation of Conflict Most of the work involved in providing an agent with a reasonable level of understanding regarding the impact of local decisions lies in computation of conflict within each agent. In this section, we give two procedures for carrying out this computation. The first takes the view that the conflict set relative to a plan fragment is the collection of sets determined by finding complements of maximal feasible sets. The second constructs a representation of conflict directly as the collection of minimal infeasible sets. Both computations yield sets that provide the same information relative to exclusion and infeasibility.

Strategy 1

For every plan fragment pf_i in pfs_A:

1. compute-maximum-compatibles(*reservation-list goals*$_A - g(pf_i)$ pf_i)
2. take the complement of each maximum compatible with respect to $pfs_A - pfs_{g_i}$ where $g_i = g(pf_i)$.

The function compute-maximum-compatibles is defined as follows:

```
compute-maximum-compatibles (reservation-list goals compatible-set)
        if (null goals)
          add compatible-set to maximum compatible sets and delete subsets
        otherwise
          for every plan fragment pf_x for g_x, the first goal in goals,
              if pf_x does not exceed resource availability based on reservation-list,
                then for every resource r_x required by pf_x,
                          add 1 to reservation-list entry for r_x
                          add pf_x to compatible-set
              compute-maximum-compatibles (reservation-list (goals - g_x) compatible-set)
```

This algorithm computes the conflict set by finding maximal compatible sets and their complements. Its complexity is bounded by

$$\mid P_A \mid * [max(\mid pfs_g \mid) * \#\text{ of resources }]^{\mid G_A \mid}$$

In fact, our experiments indicate that this expression does not represent a tight bound when the scenario represents an overconstrained situation. Since there are many fewer feasible sets when the problem is overconstrained, this is not surprising. The second procedure, given below, computes minimal infeasible sets (under the assumption that plan fragment pf_x is selected). It is not hard to see that Algorithm 2 is more efficient when the problem is underconstrained, as major portions of the algorithm are not exercised when there is no hard resource constraint to test. In the worst case, Strategy 2 is exponential in (number-of-plan-fragments * number-of-resources-known).

Strategy 2

1. For each resource, r, required by plan fragment pf_x:
   (a) for each goal, let $S_r(g)$ = {plan fragments for goal g that require resource r}

(b) let $S_r = \{S_r(g_i) \mid g_i \in G_A - g_{pfx}\}$
and $S'_r = \bigcup s$ for $s \in S_r$

(c) if $copies(r) \leq \mid S_r \mid$ then

i. define F = $\{s \mid s \subseteq S'_r, \mid s \mid = copies(r) - 1$, and $s_j, s_k \in s \Rightarrow g(s_j) \neq g(s_k)\}$

ii. let CONF(r) = $\{c \mid c = S_r - s$ for $s \in$ F $\}$

2. Construct CONF = $\{c \mid c = c_1 \cup c_2 \cup \cdots \cup c_m$ where $c_i \in$ CONF(r_i) and r_i is known to agent A$\}$

3. Conflict is represented by the collection of minimal subsets of CONF.

It is interesting to note that complements of minimal infeasible sets are *not always* identical to maximal compatible sets. The reason for this phenomenon lies in the observation that the complement of some minimal infeasible set may contain more than one plan fragment for some goal. Thus the complement of a minimal infeasible set may not itself be compatible. It is true, though, that the sets computed using both strategies do result in formation of the same exclusion sets locally, hence the propagated exclusion sets are also the same in both cases.

4.5.1.7 Interdependence Relation We address two questions of importance to distributed planners *with no global knowledge or central control*: how to achieve a maximal set of goals in an overconstrained problem and how to determine when plan execution may begin. Our solution to the first involves defining a relationship among goals called *interdependence*. We show that the set of system goals may be partitioned into subsets of interdependent goals. Each subset determines a minimal set of agents which must exchange constraint information to negotiate a maximal solution. In addition, the relation determines the minimal amount of constraint information they need to exchange to guarantee consistent decisions are made. The partition is incrementally constructed through a series of inter-agent transitive closures on the defined interdependence relation. To determine when plan execution may begin, we modify a termination detection technique used in distributed operating systems.

Our work in multiagent planning described in the previous sections has concentrated upon devising a protocol for multistage negotiation [8], generating plans in a distributed environment [39], and detecting and propagating the impact of executing local plan alternatives [9]. Kuwabara and Lesser [30] have proposed an extension to our mechanisms for propagating nonlocal impact that detects overconstrained problems not found by the original protocol. In this section we concentrate upon two remaining issues: handling overconstrained problems and determination of a system "execution condition".

When a problem is overconstrained, the overall set of system goals is not jointly feasible. Thus individual agents must determine when they should abandon achievement of one or more goals to permit achievement of a maximal number of goals in the system as a whole. The second issue, determination of a system execution condition, involves devising a mechanism whereby agents can determine when they have collectively committed to a set of mutually feasible global plans that achieve a negotiated set of goals. At this point plan execution may begin.

We begin our discussion by briefly reviewing what knowledge is available to the planner, and how this information is distributed. For a more detailed account, the reader is encouraged to see [8]. We assume that achievement of a goal in a distributed planning system involves the use of local resources which are distributed among several agents. Furthermore, we assume that there are several system goals and that as a result of distributed plan generation [39], a number of alternative global plans for each goal has been determined. Each global plan exists as a collection of plan fragments distributed among multiple agents within the system. No agent knows of all the agents that participate in a single global plan nor does any agent know exactly how many alternative plans exist for a single goal. The overall objective of a distributed planner in this environment is to efficiently allocate system resources so that a maximal set of goals may be achieved.

In our planner, each system goal is assigned to some agent which is designated as its *primary negotiator*. Each goal is also designated as one of the primary goals for its primary negotiator. The primary negotiator for a goal assumes the responsibility for achievement of that goal. Moreover, the primary negotiator is the *only* agent which can relinquish the achievement of that goal. Assigning this responsibility to a single agent (for each goal) lessens the potential for incoherent behavior due to conflicting decisions made by several agents.

4.5.1.7.1 Negotiating Overconstrained Problems Multistage Negotiation begins with each primary negotiator making a tentative commitment to a plan fragment for each of its primary goals. Agents which share resources involved in these plan fragments are notified of the tentative commitments and are requested to make tentative commitments consistent with these choices. As these agents make tentative commitments, they in turn notify and make requests of agents which share involved resources. This process continues until no further plan fragment coordination is necessary. (Coordination is no longer necessary when an agent makes a commitment to a plan fragment which does not involve any additional shared resources.) Now this "path of commitment" is followed in reverse, with each agent passing a message confirming the commitment. When the primary negotiator receives this confirmation message, it knows that the system agents have tentatively committed to a complete plan for that particular primary goal. Thus, all the component pieces of a plan have

been committed for achievement of that goal.

If an agent cannot coordinate a tentative commitment with a requesting agent (due to previous commitments), the requesting agent is notified of the failure and impact information [9, 30] is passed back. Thus, impact information is determined dynamically. As a result, one can view the process of making tentative commitments as embodying several concurrent depth first searches which have been initiated by the primary negotiators. Primary negotiators incrementally learn about mutually infeasible plans as their respective searches "block" one another. As these searches progress, the primary negotiators will be able to detect if there is *no* set of plans which allow achievement of all the existing goals. This is accomplished through mechanisms presented in[9, 30]. If such an overconstrained situation exists [1], one or more primary negotiators will have to abandon some number of their primary goals depending upon the severity of the resource constraints.

The process of selecting a maximal set of mutually feasible goals is further exacerbated by the fact that the *severity* of an overconstrained situation is itself determined dynamically. This becomes evident when one views the process from the perspective of concurrent searches. As a result of the search for mutually feasible plans, two primary negotiators may determine that they are responsible for two goals which are in fact mutually *infeasible*. That is, neither can complete a search without blocking the search initiated by the other. Through a simple negotiation, these agents can agree upon which goal is to be achieved and which is to be abandoned. Later, however, a third primary negotiator could determine that its primary goal is infeasible with the agreed upon goal, but *feasible* with the relinquished goal. Therefore, the primary negotiators must be able to alter the negotiated set of goals as new constraints are determined. Such a capability is required if a maximal set of feasible goals is to be determined.

To meet this requirement and to ensure consistent choices for goal achievement among the primary negotiators, we propose the following strategy. We define an interdependence relation $\mathcal{R}$ as follows. Goals g_i and g_j are interdependent, $g_i \mathcal{R} g_j$ if any of the following conditions hold:

$$g_i \mathcal{R} g_j \quad if \quad \begin{cases} a. & g_i = g_j, \text{ OR} \\ b. & \exists\ a\ global\ plan\ for\ g_i,\ p_i, \\ & \quad a\ global\ plan\ for\ g_j,\ p_j, \\ & \quad \ni: p_i, p_j \in \psi_{r,l}\ for\ some\ resource,\ r, \text{ OR} \\ c. & \exists\ g_k \ni: g_i \mathcal{R} g_k \wedge g_k \mathcal{R} g_j \end{cases}$$

[1] In systems with several goals and a limited supply of resources, we assume that this is a frequent occurrence.

where,[2]

$$\begin{aligned}\psi_{r,i} \equiv\ & \{p_1, \ldots, p_k\}, \\ & \textit{each plan } p_i \textit{ requires resource } r, \\ & \textit{each plan } p_i \textit{ achieves a distinct goal}, \\ & \sum_{i=1}^{k} \textit{copies of } r \textit{ required by } p_i \ \succ \\ & \quad \textit{copies of } r \textit{ available.}\end{aligned}$$

It is easy to see why we call this an interdependence relation. A goal is always interdependent with itself and two goals are interdependent if they have plans which could potentially interfere with one another due to a resource constraint. In addition, two goals are interdependent if they are each interdependent with the same goal. Since the interdependence relation $\mathcal{R}$ is an equivalence relation, it induces a partition $\mathcal{P}$ on the set of system goals.

The equivalence classes induced by the interdependence relation $\mathcal{R}$ are sets of goals having the following property. If g1 and g2 are not in the same class, then the choice for a plan to achieve g1 is totally independent of the choice for a plan to achieve g2. Thus, if an agent is not the primary negotiator for any goal in a class $\mathcal{P}_i$, it need not be involved in negotiations regarding overconstrained problems associated with goals in $\mathcal{P}_i$. Therefore, each class $\mathcal{P}_i$ of $\mathcal{P}$ identifies the *smallest* group of primary negotiators that must combine their constraint information relating a set of goals.

Goal	Plan	r1	r2	r3	r4	r5
resource count		1	2	1	1	2
g1	p11	1				
g2	p21	1	1			
g3	p31		1	1		
g4	p41			1	1	1
	p42			1	1	
g5	p51				1	1

Table 5: Partition Example

For the purpose of illustration, consider the example presented in Table 5. There are five goals, and a single global plan exists to achieve each goal except goal g4, in

[2]Note that for each resource, there is a finite collection of sets $\psi_{r,1}, \ldots, \psi_{r,n}$ defined by constraints imposed by availability of that resource.

which case there are two global plans. There are five resources: one copy each of r1, r3, and r4 and two copies each of r2 and r5.

By definition:

$$\begin{aligned}
\psi_{r1,1} &= \{p11, p21\} \\
\psi_{r2,1} &= \emptyset \\
\psi_{r3,1} &= \{p31, p41\} \\
\psi_{r3,2} &= \{p31, p42\} \\
\psi_{r4,1} &= \{p41, p51\} \\
\psi_{r4,2} &= \{p42, p51\} \\
\psi_{r5,1} &= \emptyset
\end{aligned}$$

and

$$\begin{aligned}
\mathcal{P}_1 &= \{g1, g2\} \\
\mathcal{P}_2 &= \{g3, g4, g5\}
\end{aligned}$$

The sets in this example were calculated with a global view of resource constraints to ease introduction of the concepts presented. In a distributed planner, however, no agent has a global perspective. Resource constraints are determined incrementally in a distributed manner. Thus it follows that the partition $\mathcal{P}$ induced by interdependence must be computed incrementally in a distributed manner by the primary negotiators.

When a primary negotiator, say Agent X, is notified of a goal exclusion as a result of a search it initiated, Agent X begins to construct the equivalence class associated with the excluded goal(s). Agent X performs a transitive closure of the relation $\mathcal{R}$ on the excluded goal(s) using its currently known constraints. This transitive closure returns a set of goals which are related directly or indirectly by locally known constraints. The constraints used to determine this set are transmitted to the primary negotiators for the related goals. Upon obtaining this notification, these agents also perform a local transitive closure and pass the results to the primary negotiators for the goals involved in their respective constraints. Thus, at the end of this inter-agent transitive closure, the primary negotiators for a particular equivalence class, $\mathcal{P}_i$, will have identical knowledge about the set of constraints relating their goals through the interdependence relation. Using this information, each of the primary negotiators for goals in $\mathcal{P}_i$ can determine a maximal subset of goals in $\mathcal{P}_i$ which are mutually feasible. Since we assume each primary negotiator is using the same algorithm and has the same knowledge about constraints, the results in each agent will be consistent and no acknowledgement messages are required. As new constraints are determined, this process is repeated, ensuring that decisions about *which* mutually infeasible goals

Primary Negotiator	A	B	C	D	E
Primary Goals	g1	g2	g3	g4	g5
Known Constraints			$\neg g3 \vee \neg g4$	$\neg g3 \vee \neg g4$	
Abandoned Goals			g3		

Table 6: After First Inter-Agent Transitive Closure

Primary Negotiator	A	B	C	D	E
Primary Goals	g1	g2	g3	g4	g5
Known Constraints			$\neg g3 \vee \neg g4$	$\neg g3 \vee \neg g4$	$\neg g5 \vee \neg g4$
			$\neg g5 \vee \neg g4$	$\neg g5 \vee \neg g4$	$\neg g3 \vee \neg g4$
Abandoned Goals				g4	

Table 7: After Second Inter-Agent Transitive Closure

should be abandoned are consistent and that a maximal set of goals remains to be achieved.

Using the previous example (see Table 5), consider the following scenario. Suppose that each of the five goals has been assigned to a distinct primary negotiator and concurrent searches for mutually feasible plans has begun. Table 6 shows the goal distribution, the currently discovered constraints, and the goals which have been abandoned. Notice that as the result of a previous inter-agent transitive closure, Agents C and D have determined that goals g3 and g4 are mutually infeasible and Agent C has abandoned g3 so that g4 may be achieved. At this point, suppose that Agent E is notified of a constraint relating g4 and g5, $\neg g5 \vee \neg g4$. Since this is the only constraint known to Agent E, it notifies the primary negotiator for g4, Agent D, of this constraint. Upon receiving this constraint, Agent D performs the transitive closure described and determines that goals g3, g4 and g5 are related by the constraints $\neg g5 \vee \neg g4$ and $\neg g3 \vee \neg g4$. Agent D notifies Agent C of the constraint $\neg g5 \vee \neg g4$ and Agent E of the constraint $\neg g3 \vee \neg g4$. Neither Agent C or Agent E have any further related constraints, so no new messages are passed. Using this constraint information, Agents C, D, and E can determine that goal g4 should be abandoned so that both goals g3 and g5 may be achieved (see Table 7). Notice that the information needed to resolve the overconstrained problem is the *minimal* information required to arrive at this conclusion.

4.5.1.7.2 Deciding When to Execute An important question remains. How do the individual agents know when they may execute the plans to which they have tentatively committed? Plan execution should not begin until the set of negotiated goals is guaranteed not to change *and* a set of mutually feasible plans which achieves these goals has been committed. This is referred to as the *system execution condition.* Since no single agent monitors and controls the searches of the primary negotiators, they must coordinate their activities to determine when execution may begin.

A single primary negotiator can determine when it believes the system execution condition is met by viewing its local state. When a primary negotiator determines that the system agents have tentatively committed to a global plan to achieve each of its non-relinquished primary goals, it meets the *local execution condition.* If all of the primary negotiators meet the local execution condition, then the system execution condition is met. It is important to realize that once the local execution condition is true, it does not necessarily remain true. Another search initiated by a different primary negotiator could be blocked by a confirmed tentative commitment to a plan for a local primary goal. As a result of a blocked search, impact information will be sent to the primary negotiators for each involved goal. Depending upon the nature of the resource constraints, it may be determined that both goals can be achieved, but it may be necessary for local tentative commitments to change so that a previously blocked plan may continue. If the change in commitments requires revoking a confirmed commitment, a complete tentatively committed plan for some goal no longer exists and the associated primary negotiator no longer meets the local execution condition. Clearly, this local condition could be affected by searches initiated by other primary negotiators. However, once *all* the concurrent searches for mutually feasible plans for the negotiated goals have completed, there is nothing that will change any agent's local execution condition. The solution requires a protocol whereby the primary negotiators can determine that they each meet the local execution condition and that none of the local conditions will change.

We employ a strategy similar to that used by the Gutenberg distributed operating system [46] to detect its "computation commit phase". When a primary negotiator determines that it meets the local execution condition (i.e. all its non-relinquished plans have complete tentatively committed plans), it asks all the other agents in the system if they too meet the local execution condition. If they do, this agent then notifies the others to execute their tentatively committed plans. If at least one agent has not met the local execution condition, the requesting primary negotiator continues to participate in the negotiation process without sending the execution directive. This participation may involve taking part in determining new sets of goals to achieve, answering requests about its completion, continuing its search if another agent forces it to revoke a tentative commitment, and so forth. If several agents meet the execution condition locally at approximately the same time, there may be several requests for local execution condition status in the system at the same time. This is

not a problem, however, for each of the requesting agents will reply that they do in fact meet the local execution condition and at worst, redundant execution directives are sent.

When using a strategy that "waits" for several agents to meet a condition, it is essential to guarantee that eventually all agents will indeed meet that condition. Since the process of discovering constraints and finding mutually feasible plans is bounded by an exhaustive search over a finite set of alternative plans, all agents will eventually meet the local execution condition. The system execution condition will be determined by the last agent(s) meeting the local execution condition.

4.5.1.8 Status We have discussed the problem of distributed plan generation and given a mechanism for performing this task in a class of problems in which resource allocation can be viewed as a planning problem. In distributed environments such as these, problem solving agents must cooperate to incrementally build plans to complete tasks without knowing a priori what resources are needed or how they can be utilized. Therefore, an important component of distributed plan generation involves properly assessing which local resource allocations are associated with a single global plan and which are parts of distinct global plans. Our solution to the problem requires that an agent only be aware of a limited and abstracted view of global plans.

We have also presented formalisms that permit an agent in a distributed planning system to gain knowledge about the interaction between consequences of its local actions and constraints existing elsewhere in the system. Abstractions that reflect these interactions are formulated and properties of the abstraction mechanisms are discussed. In addition, algorithms are given for computing local structures and their complexity is analyzed as an indicator of the worst case performance that can be expected. Finally, bounds on the number of transactions required to propagate local impact to distant sites are derived. We also show how this formalism provides a natural mechanism by which agents incrementally expand knowledge about the nonlocal impact of their local decisions *without* constructing a complete global view.

The formalisms discussed above have been implemented as part of a system that uses multistage negotiation [7] in distributed planning. Extensions to these formalisms that are useful in dynamic domains requiring incremental plan generation are an important area for future research.

Finally, we have addressed two questions of importance to distributed planners with no global knowledge or central control: how to negotiate overconstrained problems and how to determine when plan execution may begin. Our solution to the first involves defining a relationship among goals called *interdependence*. We show that the set of system goals may be partitioned into subsets of interdependent goals. Each subset determines a minimal set of agents which must exchange constraint information to negotiate a maximal solution. In addition, the relation determines the minimal

amount of constraint information they need to exchange to guarantee that consistent decisions are made. The partition is incrementally constructed through a series of inter-agent transitive closures on the defined interdependence relation. To determine when plan execution may begin, we modify a termination detection technique used in distributed operating systems. Implementation of these ideas is currently being incorporated into our distributed planning system. Our current planning system is implemented in Common Lisp operating on a TI Explorer. The simulation of the multiagent processing is achieved through the use of SIMULACT (described in Section 4.4.1).

In future work, we plan to address the consequences of allowing the set of system goals to change dynamically. We feel we have made a good start with our selection of a solution to determine when plan execution may begin. As discussed in [46], this solution permits the addition and deletion of system goals. Future work will address the issue of deciding how to temporarily bound the set of goals under consideration in a distributed fashion.

4.5.2 Multi-Agent Truth Maintenance System (MATMS)

During the course of normal problem solving activity, an agent, or problem solver, may formalize or make use of assumptions. Assumptions can be divided into three types: default assumptions ("unless there is evidence to the contrary, assume that the employee is getting paid"), suppositions ("suppose that the employee is not getting paid"), and new observations of the current state of the world ("the employee is not getting paid"). The common bond among the three is a *believability* which is not dependent upon any other belief.

Every inference drawn by a problem solver can ultimately be traced back to a set (or sets) of assumptions. As opposed to assumptions, the believability of an inference is dependent upon the believability of the assumptions. If an assumption set upon which an inference is based is currently believed by a problem solver, then that inference should also be believed, regardless of the type of the assumptions involved.

The assumptions and the inferences based upon these assumptions with which a problem solver is operating are referred to as the current *belief set* of the problem solver. Every time a problem solver draws an inference, makes an assumption, retracts an inference, or retracts an assumption, it changes its belief set. When a problem solver changes its belief set, many difficulties arise. How much of what was believed before the change can still be believed after the change? This is commonly called the *frame problem*. In more general terms, the frame problem is "... the inability to model side effects of actions taken in the world by making corresponding modifications in the database representing the state of the world" [3]. For example, which beliefs must be removed, and which beliefs can remain, when a particular assumption is removed?

Another problem arises when the agent introduces a belief to the knowledge base which conflicts with one which is already present. How can the intentions of the problem solver be correctly recognized? Suppose an agent wished to override a default assumption. For example, imagine that the default *is-a* attribute of each object in the knowledge base is *square*. If object RE33 is in the knowledge base, its default *is-a* attribute is *square*. If the agent realizes that RE33 is actually a circle, then *object RE33 is a circle* should be allowed to automatically override the default assumption. A belief which the problem solver has explicitly made should be allowed to override a default assumption. But what is the intended result if the problem solver introduces a belief that is inconsistent with another which is not a default assumption. If after asserting that object RE33 is a circle, the agent asserts *object RE33 is a triangle*. then the agent has explicitly made two assertions which are inconsistent. In general, when a problem solver adds a belief which contradicts an existing belief, does the belief set become inconsistent, or should the most recent belief simply override the belief with which it is inconsistent?

If a belief set of an agent does become inconsistent, classical first order logic suggests everything can be proven, and everything can be disproven, so the knowledge base is essentially worthless. It seems evident that only the attributes of those objects which are logically affected by the inconsistency should be questioned. Consider a knowledge base which contains *object RE33 is a triangle* and *object RE33 is a circle*. If there exists no logical connection between object *RE33* and object *hy77*, then the shape of object *hy77* should not be suspect.

To address problems associated with changing beliefs, truth maintenance systems [18, 14, 37, 35] have been developed for use with single problem solvers. Whenever the problem solver adds or retracts a belief, the truth maintenance system is invoked to manage the beliefs. For instance, when an assumption is removed, the system can determine (after an indeterminate amount of time) which inferences have to be removed because they depended, either directly or indirectly, on the acceptance of the assumption. In addition, if two beliefs are inconsistent, the entire knowledge base is not rendered useless. Rather, the truth maintenance system can determine which subset of beliefs in the knowledge base are inconsistent; the rest of the knowledge base remains consistent. Default assumptions are also handled appropriately. That is, default assumptions are overridden when necessary, and "come back" if a problem solver should retract the belief which overrides it. As an example, consider the case when there exists a default assumption *Car X has four wheels* and a problem solver asserts *Car X has six wheels*. If the problem solver retracts *Car X has six wheels*, then the truth maintenance system would restore the belief that Car X has four wheels.

In addition to solving these problems, the truth maintenance system records every inference made. Problem solving becomes more efficient because every inference need only be made once. Suppose a problem solver is involved in a series of long, expensive

computations. If it decides to stop the computations to perform another task, upon returning to the original task, without a truth maintenance system, the problem solver might be forced to start the task from the beginning again, thus having to recompute many results. With the truth maintenance system, the problem solver can continue essentially where it had halted. When the problem solver re-asserts the assumptions which were in its belief set while it was performing the original task, the truth maintenance system restores any inference the agent had in its belief set while performing the task.

4.5.2.1 Problem Definition Our work is concerned with a distributed problem solving environment in which there are a number of agents cooperating by passing messages to each other requesting action and by sharing inferences in a central knowledge base. Each agent's belief set is kept within the central knowledge base. When an agent adds an inference to its belief set, that inference is shared with the other agents because the validity of an inference in the central knowledge base depends only upon the validity of its preconditions. As an example, suppose the following rule exists in one of the agents:

$$A \wedge B \Rightarrow C$$

The rule can be interpreted as "If A and B are believed, then C is believed." Once the problem solver enters this knowledge into the shared knowledge base, any problem solver which believes A and B will believe C.

There are many difficulties in managing a shared knowledge base using the techniques that a conventional truth maintenance system employs to manage a knowledge base accessed by only one problem solver. Every problem present in the single agent environment is also present in the multi-agent environment, and many additional difficulties are encountered that are due to the distributed aspects of the problem solving system. First, a single agent truth maintenance system organizes its knowledge base so that the problem solver "sees" only those beliefs which are in its current belief set. A truth maintenance system in a multi-agent environment must perform this task for each of the agents. In doing so, the system must handle the possibility that one agent may believe a value for a piece of knowledge, while another may believe the opposite. To illustrate this phenomenon, consider the fact that the use of supposition by one agent should not modify the beliefs of another agent. For example, imagine that one agent believes that Resource X is available. If another agent, while engaging in hypothetical reasoning, adds the supposition that Resource X is unavailable, the first agent should still believe that Resource X is available. The truth maintenance system must also handle situations in which one piece of knowledge may be currently believed by any number of agents and at the same time disbelieved by any number of other agents.

Another difficulty arises when one realizes that the assessment of the current state of the world is achieved through the combined efforts of all agents. That is, part of each agent's task is to "fill in" the incomplete portions of an overall assessment in order to aid one another. For the most part problem solvers will agree with each other, but there will be times when two problem solvers disagree on a piece of knowledge in the knowledge base. For example, PS_1 (Problem Solver 1) might believe Resource X is available, and PS_2 might believe Resource X is unavailable. A discrepancy of this type could cause problem solving to diverge beyond the point of recovery. It is important that *inconsistencies between problem solvers' assessments of the current state of the world be recognized, and an attempt made to resolve them.* Often there will be times when the "differences of opinion" cannot be resolved. At these times, the discrepancies must be permitted to stand, hopefully to be resolved in the future.

Finally, arguments concerning the importance of efficiency in a truth maintenance system are significantly magnified when comparing a multi-agent environment to a single agent environment. The truth maintenance system managing a knowledge base shared by multiple problem solvers must be prepared to shift its focus of attention from one problem solver to another quickly, even if just to answer queries. This problem is not encountered in a single agent environment since the truth maintenance system is always concerned with the single agent. Therefore, even if the other problems in managing a knowledge base shared by multiple agents are addressed adequately, the system might be too slow to be useful in any practical problem solving system.

4.5.2.2 Summary of Results The Multi-agent Assumption-based Truth Maintenance System (MATMS) has been developed to manage a knowledge base shared by multiple problem solvers. Each problem solver has its beliefs "independently" managed in a manner similar to that provided by a conventional truth maintenance system. That is, every problem solver in the system can add and retract beliefs from its belief set and the MATMS will ensure that the belief set remains sound and complete. Every inference and only those inferences derivable from the set of assumptions in the belief set are included in the belief set. A fundamental difference in this system as opposed to conventional truth maintenance systems is that an inference provided to the MATMS is also made available to other problem solvers Any agent which believes the assumptions upon which an inference is based has the inference placed in its belief set.

The MATMS is based upon de Kleer's ATMS [14, 15] because of the ATMS's inherent adaptability to a multi-agent environment since the ATMS supports multiple belief sets at any given time. While the ATMS can be used in only a single-agent environment, the MATMS can be used in an environment involving any number of agents sharing a central knowledge base. The ATMS regards multiple derivations for

an inference and situations in which an inference replaces an assumption as unlikely events. Because such events are likely to occur in a multi-agent setting, close attention in the design of the MATMS has been given to these cases.

As is the case for some other truth maintenance systems, the MATMS supports the use of default assumptions but views default assumptions in an unconventional manner. Much of the MATMS's efficiency results from our treatment of default assumptions as constant entities. The "default knowledge base" does not change much during the course of problem solving. A problem solver's belief set can therefore be characterized as the default knowledge base with an "overlay" placed upon it. The overlay blocks certain defaults and includes knowledge for which there is no default. The MATMS is efficient largely because it focuses its efforts on managing these overlays, not the entire belief set of an agent. By concerning itself only with the overlays, the MATMS can switch from addressing one problem solver's belief set to addressing another's expeditiously. It can also change an individual problem solver's belief set quickly, because the default knowledge is not explicitly carried over from one belief set to another.

The most important feature of the MATMS is that it provides the foundation for resolving inconsistency between agents, while supporting the notion that two problem solvers can have different views concerning the state of a particular piece of knowledge. The MATMS handles differing views by allowing independent belief sets for each of the agents. It supports resolving inconsistency between agents by providing a mechanism for comparing two agents' belief sets. Comparison is swift because the way the MATMS compares belief sets is by comparing the overlays. The overlays do not include the default knowledge, which is usually the largest portion of an agent's belief set. Any problem solver can ask the MATMS to compare the beliefs of any agents in order to recognize discrepancies. The best manner in which to resolve the inconsistencies is a matter for future research.

4.5.2.3 Multiagent Environment The MATMS has been designed to operate in the context of a distributed knowledge based system for managing a large communications system. The knowledge based system architecture consists of multiple, cooperative, distributed agents. The functionally specialized agents at the local level are Performance Assessment (PA), Fault Isolation (FI), and Service Restoral (SR). An important feature of the system is the cooperation of the agents. Cooperation at the local level is by two methods. First, problem solvers cooperate by coordinating their actions. An agent may request another to perform some task to further the overall problem solving. This is achieved through an exchange of messages. The other mechanism for cooperation is through sharing knowledge concerning the current state of the communications network. Inferences of one agent are shared with the others, in a central knowledge base. The shared knowledge base is managed by the Knowl-

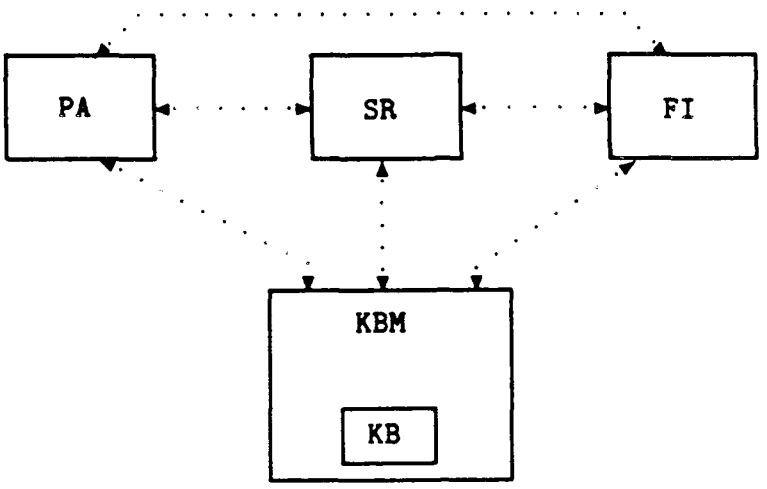


Figure 19: Local System Architecture

edge Base Manager (KBM). The local system architecture is shown in Figure 19 (the dotted lines indicate the interagent communications path).

The KBM has the responsibility for managing the knowledge base. Within the KBM, some type of truth maintenance system must be active. Its tasks involve regulating each problem solver's beliefs in a manner similar to that provided by conventional truth maintenance systems, as well as providing a means by which an agent can easily share its inferences with the others. While the truth maintenance system will not directly address resolution of inconsistencies across agents, it must provide an efficient mechanism by which the KBM can recognize inconsistencies between PA, FI, and SR.

Prior work on truth maintenance systems have not resulted in the development of mechanisms that support activity in a multi-agent environment such as this. Existing truth maintenances systems have been designed for single agent systems and have not had to deal with some of the issues that arise in multi-agent systems.

4.5.2.4 Existing Truth Maintenance Systems Existing truth maintenance systems have failed to address the need for a system in which multiple problem solvers' inferences are controlled by a single truth maintenance system. However, because we have adopted much of the terminology of conventional truth maintenance systems, and the MATMS borrows heavily from concepts developed in existing truth maintenance systems, the two dominant classes of truth maintenance systems are presented.

Doyle's Truth Maintenance System (TMS) [18] was the first domain independent truth maintenance system. Doyle proposed that reasons for believing or using each belief, inference rule or procedure be recorded. This allows new information to displace previous conclusions and a consistent knowledge base to be kept. In the TMS, each belief in the knowledge base is explicitly marked as either IN or OUT, where IN means that the belief has at least one currently acceptable reason, and OUT means that it has no currently acceptable reason for belief. Given a belief or a justification for an existing belief, the job of the TMS is to determine the belief status of each of the beliefs in the knowledge base, thus retaining one consistent knowledge base.

Doyle's TMS defined the class of *justification based* truth maintenance systems. That is, the status of each belief is determined by searching through each justification until reaching a set of assumptions. If the set of assumptions are valid (or believed), then the belief is valid. Considering that these justifications are examined for each belief in the knowledge base, and any particular chain of inferences which eventually leads to a particular inference in question may be long, updating the knowledge base may require a long period of time.

The TMS was deemed inappropriate for a multi-problem solver environment because it maintains only one belief set at a time. Given any point in time, the TMS has one set of beliefs which are IN and one set which are OUT. Switching belief spaces is cumbersome because the status of each belief has to be directly recomputed. As we have observed, in a multiple problem solver environment, the truth maintenance system must be able to switch belief spaces quickly.

De Kleer, in his Assumption based Truth Maintenance System (ATMS) [14, 15], recognized this problem also, though not for the same reasons. De Kleer was interested in hypothetical reasoning, in which assumptions are made often, and results compared against the assumptions. Therefore, the ATMS was designed explicitly to switch belief sets efficiently. The ATMS is the foundation for the MATMS, and as such will be discussed in much greater detail.

In order to create a system which could switch beliefs sets quickly, de Kleer recognized that an inference is ultimately dependent on a set (or sets) of assumptions. That is, an inference may be derived from other inferences, and these inferences may have been derived from other inferences, but eventually this trace will find assumptions only. Therefore, when a problem solver changes its belief set, the justifications of the inferences in the previous belief set do not have to be traced to determine if they still have valid support. Rather, each inference could be tested to see if the assumptions upon which it is based are still present in belief set.

In the ATMS the entire set of beliefs is divided into sets called *contexts*; each context represents a belief set. Essentially a context is defined by its assumption set, which is called an *environment*, and includes all inferences which can be derived, either directly or indirectly, from the environment.

As a context is associated with a particular set of beliefs, each belief is associated with a list of contexts to which it belongs. Much of the ATMS's work involves ensuring that each belief's label, the set of environments from which the node is derivable, is consistent, sound, complete, and minimal with respect to the justifications. A label is consistent if each environment in the label is consistent, sound if every environment can derive the belief, complete if every way to derive the belief is included in the label, and minimal if no environment in the label is a superset of another in the label. Labels must be kept this way primarily for efficiency.

There are three features which make the ATMS more appropriate than the TMS for the multi-agent system described. First, the ATMS maintains more than one belief set at a time by maintaining multiple contexts. Each agent in a multi-agent system could conceivably be operating with a different set of beliefs, so it is essential that a truth maintenance system handling their beliefs have the ability to maintain multiple belief sets. Second, a problem solver using the ATMS can change its belief set much more swiftly than if it were using the TMS, because the TMS is often forced to perform costly tracing in order to reassign belief status to each of the beliefs in the knowledge base. Switching belief sets is also quicker in the ATMS because the new belief set could already be defined. For example, suppose a problem solver utilizing the ATMS adds assumption X to its belief set. After the ATMS calculates the problem solver's new context, the problem solver retracts assumption X. When this happens, the ATMS simply returns the problem solver to its previous context. The TMS in this situation would have to reassign belief status to each belief in the knowledge base, only to return the problem solver to its original belief set. The third reason is that the ATMS handles multiple derivations for an inference better than the TMS. (Although it still does not handle it very elegantly, it is still far more capable than the TMS.) In a multi-problem solver environment, where inferencing schemes are numerous, a belief is likely to be inferred from more than one set of beliefs. A truth maintenance system in a multi-agent system must be able to determine that, if support for an inference is removed, there may be other support which keeps the belief in the current context.

The major drawback to the assumption based systems is the computation of support for each inference in the knowledge base. When an inference is added to the justification based system, only the immediate preconditions of the inference are recorded (this is what makes the system *justification based*). Therefore, adding inferences is not difficult. In an assumption based system, the immediate preconditions must be traced until assumption sets are found. In situations involving inferences already in the knowledge base that are themselves immediate preconditions to other beliefs, this causes inefficiency. Each belief which is either directly or indirectly influenced by a "new" inference must have the assumption set upon which it is based recomputed. Therefore, adding inferences might require a significant amount of computation.

An interesting observation is how each truth maintenance system handles default assumptions. The focus of each is how to make default assumptions "come back" when a belief which previously overrode the default is retracted. Both the TMS and the ATMS have chosen to include default assumptions explicitly in the belief set of the problem solver. If the number of default assumptions is large, then each system is hampered.

Martins and Shapiro in [35] present a similar comparison of assumption based and justification based truth maintenance systems. They also present a useful example of how an assumption based system manages beliefs as opposed to a justification based system.

4.5.2.5 MATMS Design The MATMS has been designed for use in a system involving any number of agents sharing a central knowledge base. In such a system, each problem solver registers its beliefs with the MATMS. An inference is registered along with the beliefs upon which it directly depends, and the job of the MATMS is to maintain multiple belief sets. Thus the MATMS is responsible for placing an inference in any belief set which contains the assumptions upon which it is based, informing an agent when its belief set becomes inconsistent, changing the belief set of an agent efficiently, and switching its focus of attention from one agent to another quickly. In this section, the definitions, data structures, algorithms, and general operations of the MATMS are discussed.

4.5.2.5.1 Definitions As a matter of convention, the operators of propositional logic are utilized from this point on. Specifically, the logical connectives of interest are $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implication), $\neg$ (not), and $\perp$ (false). Some examples are $A \Rightarrow B$ (A implies B), $C \wedge D \Rightarrow \perp$ (the quantity C and D implies false), and $E \vee F \Rightarrow \neg G$ (E or F imply not G). Shorthand notation will be used for $\wedge$: $C \wedge D \Rightarrow \perp$ will usually be written as $CD \Rightarrow \perp$, and $((A)(B))$ means $(A) \vee (B)$.

A *proposition* is the MATMS datum that represents a piece of knowledge which a problem solver has told the MATMS. Each proposition is unique. Example propositions are "Jim is a golfer," "Radio R1 has failed," and "The Celica is being repaired."

Each proposition is attached to a *belief*, the basic datum on which the MATMS operates. Beliefs are explicitly divided into two categories: *assumptions* and *inferences*. An *inference* is a belief whose validity depends upon other beliefs. For instance, if an agent has the rule $(A \wedge B) \vee (C \wedge (E \vee F)) \Rightarrow G$ and tells the MATMS that it believes G because of it believes C and E, then G is an *inference* because it is only believed in this case if C and E are believed. An *assumption* is a belief whose validity does not depend upon the acceptance of any other belief. Assumptions are divided into three types: default assumptions, suppositions, and new observations concerning the state of the world. The last two types will be referred to as *non-default assumptions*.

For discussion purposes, we represent inferencing mechanisms as rules, but they do not necessarily have to be interpreted in a strict sense. For instance, $A \Rightarrow B$ is merely meant to represent that inference operations can be activated in the presense of A to draw the logical conclusion B. The exact meaning of an inference rule can be interpreted as "in the presense of certain beliefs, another belief is implied, no matter which other beliefs are present." In other words, if $A \Rightarrow B$, B is included in any set which contains A, such as (AB) and (AEF).

A *justification* for a belief is the set of beliefs which must be present for its validity. An assumption has no justification (the justification is nil), whereas an *inference* must have at least one justification, and may have many. The justification for an inference is comprised of two components: its immediate preconditions (the beliefs from which it can directly be inferred), and the assumptions upon which it is based (the assumptions that it ultimately depends upon). For example, considering $A \Rightarrow B$, the assumptions that B is based upon are the same as its immediate preconditions, i.e. $((A))$. If $B \Rightarrow C$, the assumptions that C is based upon are again $((A))$, but the immediate preconditions necessary for its derivation are $((B))$. If $C \Rightarrow D$ and $E \Rightarrow D$, the assumptions that D is based upon are $((A)(E))$ and the immediate are $((C)(E))$. (Recall that $((C)(E))$ should be interpreted as $(C) \vee (E)$.)

At any point in time, each problem solver has a belief set which is a set of assumptions and inferences which have been derived from those assumptions. Some of the beliefs are default assumptions, some are non-default assumptions, and some are inferences. An *environment* is a unique set of non-default assumptions under which a problem solver has operated. Environments are created incrementally so that whenever a problem solver retracts or adds a non-default assumption, an environment is created if one does not exist that matches the new set of assumptions.

A *context* is an environment and all inferences which have been derived from the environment; hence it is a group of beliefs. For every environment, there is exactly one context, and a context is created each time an environment is created. If a problem solver has never worked with a particular grouping of assumptions, the MATMS does not have this set of assumptions listed an an environment, so there is no context for this group.

A *premise* is a rule which states that a set of propositions are inconsistent. Beliefs with these propositions are therefore inconsistent. Examples of premises are:

$$\neg(\text{"Fred is dead"}\ \text{"Fred is alive"})$$
$$\textit{or}$$
$$\neg(\text{"a man is working"}\ \text{"a man is resting"})\ .$$

As is evident from these examples, premises can be specific or general. It is important to observe that a premise has no meaning until a belief is supplied to the knowledge base which has as a proposition one of the propositions named in the premise. A

set of beliefs is said to be *contradictory* if the propositions of the beliefs violate any premise. The set will be referred to as an incompatible belief set, or *incompatible.*

A context is *inconsistent* if it contains contradictory beliefs. In other words, the context is inconsistent if any subset of its beliefs is an incompatible belief set.

The MATMS monitors the belief set of a problem solver by recording the context in which the problem solver is currently working. A problem solver is working in a particular context if it has explicitly told the MATMS that it holds all of the assumptions defining the context.

A problem solver *retracts* an assumption when it asks the MATMS to remove the assumption from its current belief set. Note that neither the assumption is actually removed from the knowledge base, nor the inferences which have been registered as depending upon it. The problem solver is just placed in a new context. Thus, the problem solver *switches contexts* whenever it adds or deletes an non-default assumption.

4.5.2.5.2 Data Structures The MATMS is a frame-based system in which there are five basic types of objects: beliefs, inferences, assumptions, contexts, and incompatibles. Our discussion of the data structures of the MATMS begins with belief. Each belief has slots *proposition*, *contexts in*, and *influences. Contexts in* is a list of contexts in which the belief holds. *Influences* is a list of beliefs which this belief directly influences. The frame for belief, as well as the other frames, is depicted in Figure 20.

The beliefs, as previously mentioned, are explicitly divided into two classes at any point in time: assumptions and inferences. Each class inherits from the belief frame. The inference class, however, also includes the slots *assumptions based upon* and *immediate preconditions. Immediate preconditions* is a list of sets of beliefs from which inference rules were applied to produce the resultant inference. Each belief in each of these sets has this resultant inference as a member of its *influence* slot. If the length of *immediate preconditions* is greater than one, multiple derivations for the inference have been provided to the MATMS. *Assumptions based upon* are the minimal[3] sets of assumptions from which the inference has been derived.[4] If the environment of a context is a superset of any set in *assumptions based upon*, then the inference is included in the context. *Assumptions based upon* is constructed by tracing the chain of *immediate preconditions* until assumptions are found.

The frame for a context has *environment*, *inferences*, and *incompatible belief sets*

[3]Minimal in terms of set inclusion. For instance, the minimal sets of $((AB)(ABC)(DE))$ are $((AB)(DE))$. (ABC) is not included because it is a proper superset of (AB).

[4]The *immediate preconditions* slot is not minimal because it is necessary to maintain records of every way the inference has been derived in case a derivation is retracted by the problem solver. This is discussed in "Write Operations."

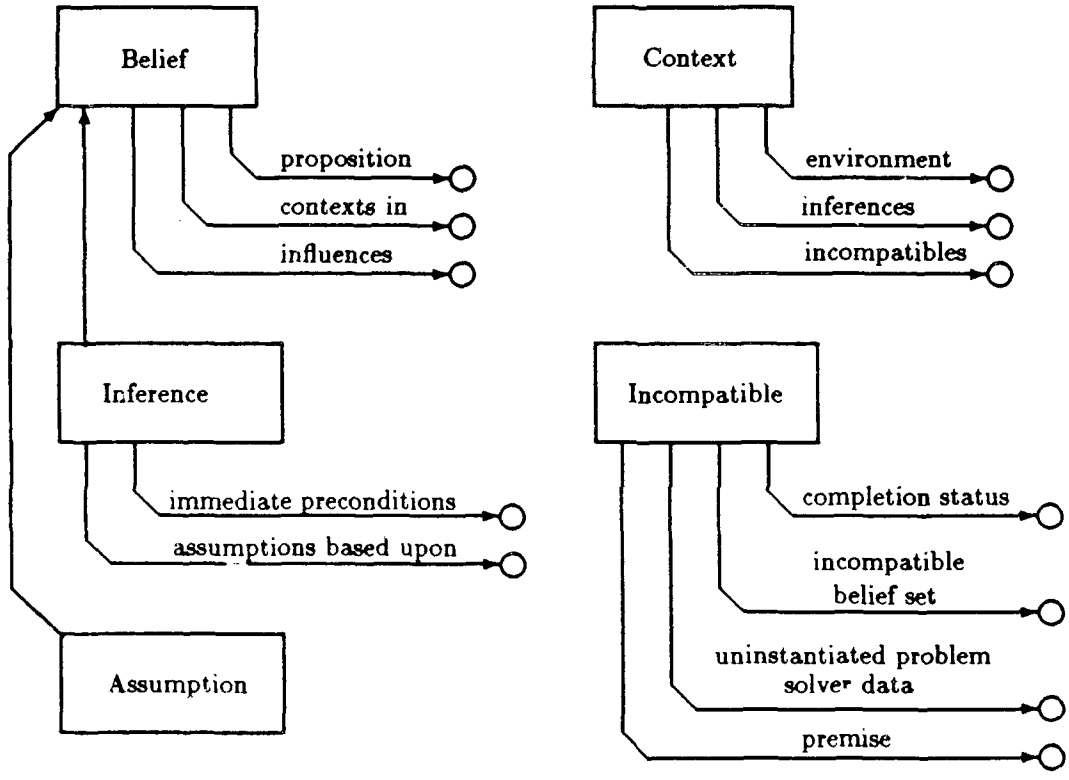


Figure 20: Net Connecting MATMS Frames

slots. *Environment* is the unique set of non-default assumptions which defines the context. *Inferences* is the list of all inferences which have been derived from the environment. An inference is included in *inferences* if it is based upon at least one set of assumptions of which at least one is non-default, and all of the non-defaults are included in *environment*. The *incompatible belief sets* slot contains a list of sets of beliefs in the context which have been previously defined in a premise as being incompatible. Each belief in *incompatible belief sets* is a member of *environment* or *inferences*. By definition, if *incompatible belief sets* is not empty, the context is inconsistent.

At this point, it is appropriate to present an example to illustrate the data structures. Imagine a simple two-agent system consisting of Agent_1 and Agent_2, and suppose that part of each agent's task is to plan the schedule and activities of students. The following scenario occurs:

1. Initially, each agent is working with only the default assumptions. This "null" context will be referred to as *C0*. (Alternative symbolic representations are given so that a interpretable table can be presented at the end.)

2. Agent_1 adds the assumption "Economics 337 will be held in room 445 of Hamilton Hall 6/5/88" (*A1*) to its belief set. (There is a default assumption which states that Economics 337 is usually held in Morley Hall.) This causes the MATMS to create a new context, *C1*. Agent_1 is then placed in *C1*.

3. Agent_1 adds the assumption "Hamilton Hall is further from the dorms than Morley Hall" (*A2*) to its belief set. This causes the MATMS to create a new context, *C2*. Logically, Agent_1 is then placed in *C2*.

4. Agent_1 adds the inference "It will take longer than normal to go to class tomorrow" (*I1*) to its belief set. The inference is based upon "Economics 337 will be held in room 445 of Hamilton Hall tomorrow" (*A1*) and "Hamilton Hall is further from the dorms than Morley Hall" (*A2*). The MATMS adds the inference to every context which contains *A1* and *A2*; in this case, only *C2*.

5. Agent_1 adds the inference "A person in Economics 337 should leave early for class tomorrow" (*I2*) to its belief set. The inference is based upon only "It will take longer than normal to go to class tomorrow" (*I1*). The MATMS adds the inference to every context which contains *I1*; again only *C2*.

6. Agent_2 adds the assumption "Economics 337 will be cancelled tomorrow" (*A3*) to its belief set. This causes the MATMS to create a new context, *C3*. Agent_2 is then placed in *C3*.

7. Agent_2 adds the inference "A person in Economics 337 should play golf tomorrow" (*I3*) to its belief set. The inference is based upon only "Economics 337

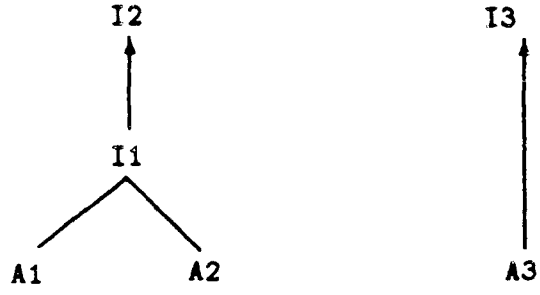


Figure 21: Inference Tree for Example Illustrating Data Structures

will be cancelled tomorrow" (*A9*). The MATMS adds the inference to every context which contains *I1*; in this case, only *C3*.

Figure 21 shows the inference tree for this knowledge base, and the following table represents the data structures of the MATMS at this point in problem solving. In the table, "assumptions" is shorthand for "assumptions based upon," and "preconditions" is short for "immediate preconditions."

	C0	C1	C2	C3
environment	()	(A1)	(A1 A2)	(A3)
inferences	()	()	(I1 I2)	(I3)
incompatibles	()	()	()	()

	A1	A2	A3	I1	I2	I3
contexts in	(C1 C2)	(C2)	(C3)	(C2)	(C2)	(C3)
influences	(I1)	(I1)	(I3)	(I2)	()	()
assumptions	*	*	*	((A1 A2))	((A1 A2))	((A3))
preconditions	*	*	*	((A1 A2))	((I1))	((A3))

Continuing with the discussion of the data structures, as noted in the previous section, premises may or may not be used by MATMS. For example, consider a premise "a man cannot be working and resting at the same." If a problem solver never supplies a proposition pertaining to a particular man and his work status, then this premise will never be used. But suppose a problem solver supplies "Jim is working" after supplying "Jim is a man." The MATMS must recognize that "half" of the premise has been supplied. If the other half is supplied, e.g. "Jim is resting", then the instantiation of the premise will be complete; the beliefs representing "Jim

is working" and "Jim is resting" form an incompatible. Note that this premise can be instantiated many times.

The data structure for *incompatible* is used to capture this notion of how incompatible belief sets are created. The slots are *completion status*, *incompatible belief set*, *uninstantiated problem solver data*, and *premise*. *Completion status* can have either of two values: complete or incomplete. A status of *INCOMPLETE* means that only a subset of the propositions involved in a premise have been proposed by problem solvers. This would be the case in the above scenario right after a problem solver supplies "Jim is working." *Uninstantiated problem solver data* refers to the data mentioned in the premise which have not yet been supplied by a problem solver. An incompatible which has completion status of *COMPLETE* details a complete set of beliefs which cannot exist in the same context. This set is the *incompatible belief set*. As an example, the following incompatible will become complete when (and if) a problem solver provides the MATMS with an belief whose proposition is "Fred is asleep". In this case it is important to differentiate between a problem solver datum and the MATMS belief which represents it, so we use the notation B(x) to mean "the belief representing the problem solver datum x".

completion status:	INCOMPLETE
incompatible belief set:	(B("Fred is awake"))
uninstantiated problem solver data:	("Fred is asleep")
premise:	¬("Fred is awake" "Fred is asleep")

When and if the incompatible becomes completed, all contexts will be searched to determine if any one of them includes the incompatible set. The completed frame would look like:

completion status:	COMPLETE
incompatible belief set:	(B("Fred is awake") B("Fred is asleep"))
uninstantiated problem solver data:	()
premise:	¬("Fred is awake" "Fred is asleep")

The details associated with instantiating and using incompatibles will be discussed more completely in subsequent sections.

4.5.2.5.3 Write Operations Operations of the MATMS will be discussed from the perspective of the MATMS. The next two sections detail how the MATMS manipulates its data structures in response to problem solver requests.

Problem Solver Proposes Adding Assumption:

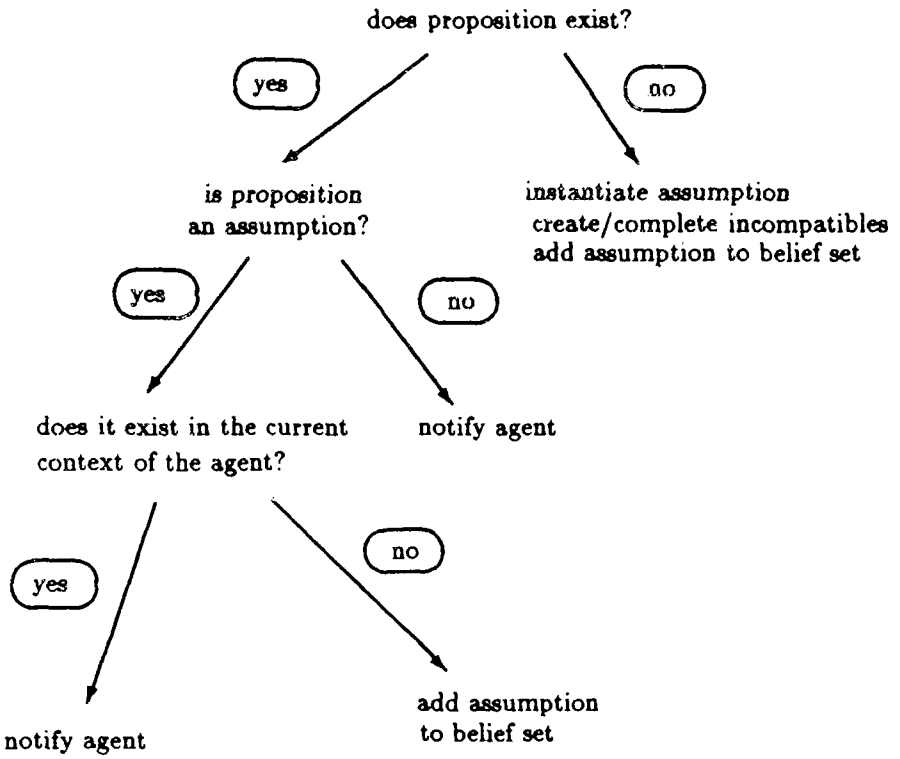


Figure 22: Decision Tree for *Problem Solver Proposes Adding Assumption*

There are four basic write operations: a problem solver proposes adding an assumption to its belief set, a problem solver proposes removing an assumption from its belief set, a problem solver proposes an inference, and a problem solver proposes removing a particular justification for an inference. Each is discussed in detail.

The MATMS follows the operations described below and in Figure 22 when a agent proposes adding an assumption to its belief set. Note that implicitly an agent may only request to add a non-default assumption to its belief set. This will be discussed in a later section.

When a problem solver proposes adding an assumption to its belief set, the MATMS first determines if a proposition is already present which matches the proposed assumption. If the proposition already exists, then there must exist either a default assumption, a non-default assumption, or an inference which has the proposition in its *proposition* slot. If it is a default assumption, the agent is not operating properly, because a problem solver cannot "re-accept" a default assumption by explicitly attempting to add it to its belief set. If it is a non-default assumption, the

MATMS must check to see if it is already present in the problem solver's current context. If it does, then the problem solver clearly made a mistake and is so notified. If an inference has the proposition in *proposition*, then the problem solver is notified that it is attempting to assume something which has already been derived.

If a proposition did not already exist, one is instantiated at this time. Whenever a new proposition is made, the incomplete incompatibles are examined to determine if the new problem solver datum will complete any of them. Then the premises are searched to determine if any new incompatibles should be started. After this search, the assumption is instantiated with the new proposition as its *proposition*.

At this point, if there is an assumption with the proposition as its *proposition*, and that assumption is not already in the problem solver's belief set, the MATMS must find or create a context which has an environment containing only the environment of the old problem solver context and the new assumption. Note that this has no effect on the belief sets of the other problem solvers; they remain in their current contexts.

The creation of a context occurs in four phases:

1. The context is first instantiated with most of the slots unfilled, and only the *environment* slot is set, with a list including the new assumption and the assumptions of the previous context.

2. Each inference which has been derived from the set of assumptions is now placed in the *inferences* slot of the context.

3. All incompatibles are now examined to see if the new context is inconsistent. If it is, the *incompatibles* slot is set appropriately.

4. The context is appended to the *contexts in* slot of each inference and assumption included.

Whether or not a context had to be created for this different environment, the problem solver now switches contexts. If the context is inconsistent, the problem solver making the assumption must be notified. A list of incompatible beliefs and how to remove each belief (this can only be done by retracting assumptions) are returned to the problem solver.

Problem Solver Proposes Removing Assumption:

Figure 23 illustrates the procedure the MATMS follows when a problem solver asks the MATMS to remove an assumption from its current belief set. Again, recall that a problem solver may only ask to remove a non-default assumption from its belief set.

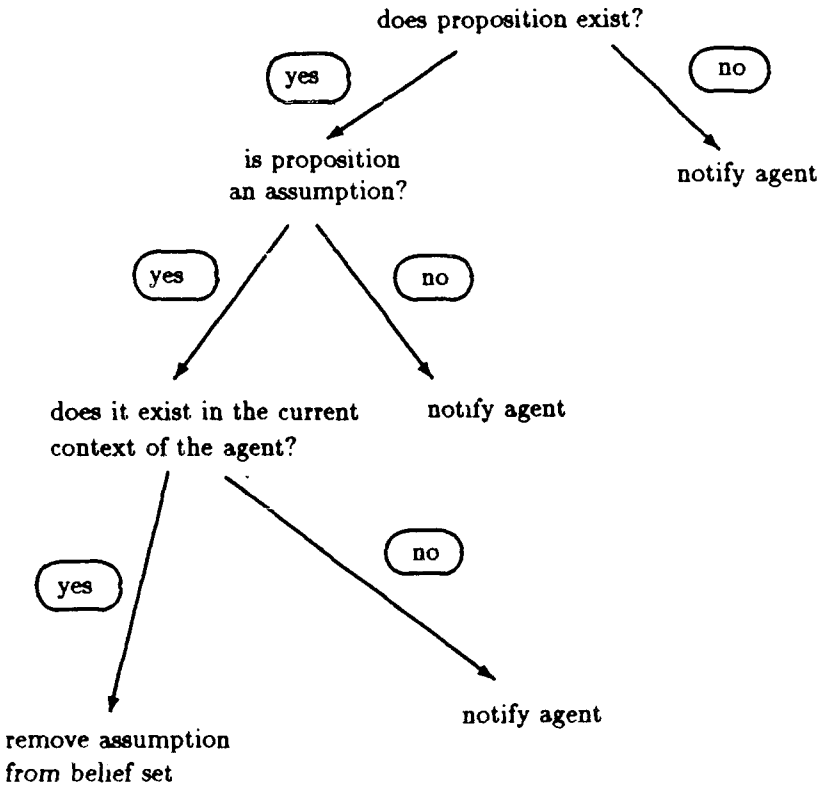


Figure 23: Decision Tree for *Problem Solver Proposes Removing Assumption*

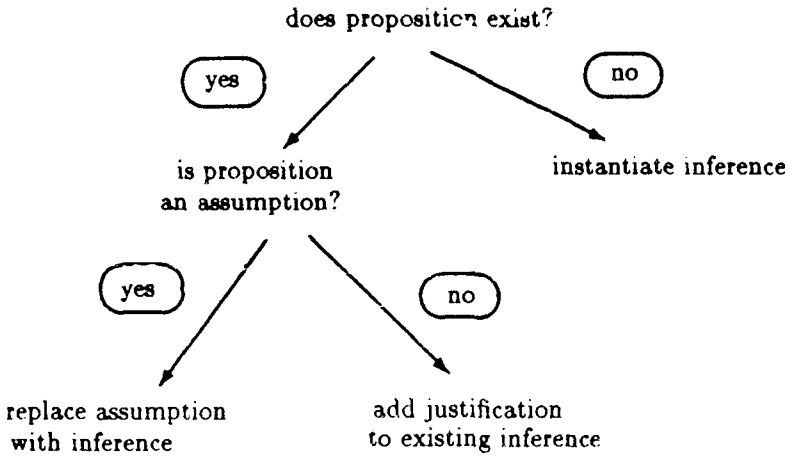


Figure 24: Decision Tree for *Problem Solver Proposes Inference*

When a problem solver asks to remove a particular assumption from its belief set, it is asking to be placed in a context which includes all assumptions of its current environment except for the assumption in question. Clearly the assumption must already must be present in the MATMS knowledge base, and specifically it must be in the present context of the problem solver. The problem solver is notified accordingly if this is not the case.

To actually remove an assumption from a problem solver belief set, a context is sought whose environment matches the new set of assumptions. If it is not found, then it is created by the procedure described in the previous section.

The problem solver is then placed in the new context. If the context is inconsistent, the agent is notified.

Problem Solver Proposes Inference:

As with other operations discussed, the proposition which corresponds to the inference supplied by the problem solver is the key for how the MATMS decides on an action to take. If the proposition is not present, clearly the inference must simply be instantiated. If the proposition exists and is attached only to an assumption, the inference must be instantiated. If an inference is present which has the proposition as its *proposition*, the problem solver is proposing what it believes is a valid justification for that inference, whether or not it realizes that the inference already exists. The decision tree implied here is depicted in Figure 24.

To clarify the discussion, registering an inference with the MATMS will be divided into three types of operations: instantiating a new inference, replacing an assumption with an inference, and supplying an existing inference with another assumption.

Common to all three types of inference operations is a trace of the justification supplied. When an agent proposes an inference, it also provides the justification for the inference. The first action the MATMS takes is to trace each belief in the justification until the assumptions upon which the belief is based are found. The minimal combinations of these assumption sets are the assumptions upon which the inference is based. For example, suppose we have the following knowledge base for a two-agent system: "Route 101 is fast" because "there aren't many policemen on Route 101"; "Route 101 is fast" because "Route 101 is a four lane highway"; and "Route 56 is slow" because "there are many potholes on Route 56". An agent then proposes the inference "Route 101 is preferred over Route 56" with justification ("Route 101 is fast" "Route 56 is slow"). The minimal subsets upon which the inference would be based are: (("there aren't many policemen on Route 101" "there are many potholes on Route 56") ("there are many potholes on Route 56" "Route 101 is a four lane highway")).

Adding a new inference to the MATMS knowledge base is the most straightforward of the three types of operations. The procedure for adding an inference is to instantiate the inference with the proposition as its *proposition*, the minimal assumption sets determined above as its *assumptions based upon*, and the justification itself as its sole *immediate preconditions*. After the inference is instantiated, the inference is used in an attempt to complete the existing incompatibles. Also, new incompatibles are created from relevant premises.

The more complicated steps in adding a new inference pertain to contexts. The *assumptions based upon* slot determines to which contexts the inference should be added. For each assumption set in *assumptions based upon*, the intersection of each assumption's *contexts in* slot is taken. This list represents the list of contexts to which all assumptions in the set belong. The inference is added to the *inferences* slot of each context in this list, and the context is added to the inference's *contexts in* slot.

Note that when a problem solver adds an assumption to its belief set, no more than one context can be found inconsistent *as a direct result of adding the assumption*. Only the problem solver which added the assumption might be placed into an inconsistent context. However, when a problem solver registers an inference with the MATMS, many contexts might be found inconsistent. This implies that other problem solvers might suddenly be working with inconsistent belief sets as a result of one problem solver registering an inference. Each problem solver whose belief set becomes inconsistent must be notified and corrective alternatives must be supplied.

An inference can replace an assumption when a problem solver has derived something which either it or another problem solver had previously assumed. (This as-

sumption could be either default or non-default.) This is a likely occurrence as a result of "normal" problem solving activity. As a problem solver proceeds, it may reach a point at which it does not know the present (or any reasonable) value for a particular piece of knowledge necessary to continue working, so it "guesses" a value. Later either it or another problem solver may produce or recognize confirming evidence, which in effect replaces the assumption with an inference. Conceptually, replacing an assumption with an inference involves replacing all occurrences of the assumption in the knowledge base with the inference.

The first step in replacing an assumption in the MATMS knowledge base with an inference is to instantiate the inference using the procedure described earlier in this section. Instantiation is independent of the fact that the inference will be used to replace an assumption.

No new incompatible will be created nor will any be completed when an inference replaces an assumption, because incompatibles are ultimately dependent upon propositions, not beliefs. An assumption with this proposition as its *proposition* has already been created, so the incompatibles which are relevant to the problem solver datum in question have already been created or completed. They must be altered, though, because they refer to the assumption rather than the inference. All incompatibles which mention the assumption therefore have the assumption replaced by the inference.

Next, the *influences* slot of the inference must be adjusted to reflect the *influences slot of the assumption*. Each inference which the assumption influenced must have its *immediate preconditions* and *assumptions based upon* recalculated. In general this could cause a fair amount of updating if the assumption influenced many inferences.

If the assumption was non-default, the contexts which contained the assumption must be killed, because they will never be referenced again. Killing a context means removing the context from *contexts in* of each belief in the context and deleting the instantiation of the context. If a problem solver is currently working in a context which is about to be killed, then it should be notified properly. The problem solver will be told that one of its assumptions has been replaced by an inference, and generally the problem solver will simply choose to accept the assumptions on which the inference is based. This way, the problem solver's belief set will continue to contain at least the same beliefs as its original belief set.

When an inference replaces an assumption, the MATMS records the details of the transaction so that it knows what to restore in case the inference is retracted.

An example illustrates how the MATMS operates to replace an assumption with an inference. Consider a simple two-agent system consisting of Agent_1 and Agent_2. The following transaction occurs:

1. Initially, each agent is working only with the default assumptions. This "null" context will be referred to as *C0*.

2. $Agent_1$ adds the assumption "there are many potholes on Route 56" (*A1*) to its belief set. This causes the MATMS to create a new context, *C1*. $Agent_1$ is then placed in *C1*.

3. $Agent_1$ adds the inference "Route 56 is slow" (*I1*) to its belief set. The inference is based upon only "there are many potholes on Route 56" (*A1*). The MATMS adds the inference to every context which contains *A1*; in this case, only *C1*.

4. $Agent_1$ adds the assumption "Route 101 is fast" (*A2*) to its belief set. This causes the MATMS to create a new context, *C2*. $Agent_1$ is then placed in *C2*.

5. $Agent_1$ adds the inference "Route 101 is preferred over Route 56" (*I2*) to its belief set. The inference is based upon "Route 101 is fast" (*A2*) and "Route 56 is slow"(*I1*). The MATMS adds the inference to every context which contains *A1* and *A2*; in this case, only *C2*.

6. $Agent_2$ adds the assumption "Route 101 is a four lane highway" (*A3*) to its belief set. This causes the MATMS to create a new context, *C3*. $Agent_2$ is then placed in *C3*.

7. $Agent_2$ adds the inference "Route 101 is fast" (*I3*) to its belief set. The inference is based upon only "Route 101 is a four lane highway" (*A3*). The inference is added to each context which contains *A3*; in this case, only *C3*. Because the inference *I3* replaces the assumption *A1*, all contexts which include *A1* must be killed; these are *C1* and *C2*. $Agent_1$ is told that an assumption in its belief set, "Route 101 is fast", is being replaced by an inference. In order to continue with its activity, it should accept the assumption "Route 101 is a four lane highway".

8. $Agent_1$ adds the assumption "Route 101 is a four lane highway" (*A3*) to its belief set. This causes the MATMS to create a new context, *C4*. $Agent_2$ is then placed in *C4*.

Figure 25 shows the inference tree for this knowledge base. The following table represents the data structures of the MATMS at this point in the problem solving.

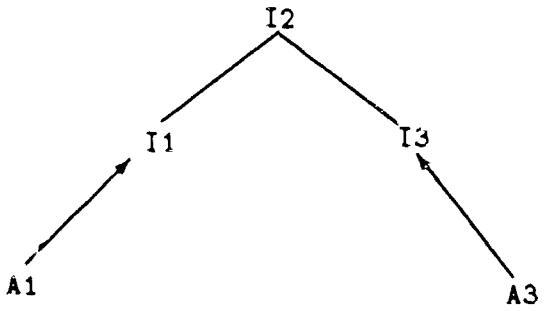


Figure 25: Inference Tree for *Inference Replaces Assumption* Example

	C0	C3	C4
environment	()	(A3)	(A1 A3)
inferences	()	(I3)	(I1 I2 I3)
incompatibles	()	()	()

	A1	A3	I1	I2	I3
contexts in	(C4)	(C3 C4)	(C4)	(C4)	(C3 C4)
influences	(I1)	(I3)	(I2)	()	(I2)
assumptions	*	*	((A1))	((A1 A3))	((A3))
preconditions	*	*	((A1))	((I1 I3))	((A3))

The algorithm for adding a justification is not very different than the one for replacing an assumption with an inference. To add a justification to an existing inference, the *assumptions based upon* of beliefs "above" the inference in the tree must be recalculated. In addition, beliefs below the inference could require updating in certain cases. Consider an additional step in the two agent system scenario presented a few paragraphs above.

9. $Agent_2$ adds the assumption "There aren't many policemen on Route 101" (*A4*) to its belief set. This causes the MATMS to create a new context, *C5*. $Agent_2$ is then placed in *C5*.

10. $Agent_2$ adds another justification for the inference "Route 101 is fast" (*I3*). The justification is only "There aren't many policemen on Route 101" (*A4*). The inference is not added to any contexts because the only context which includes *A4* already includes the inference.

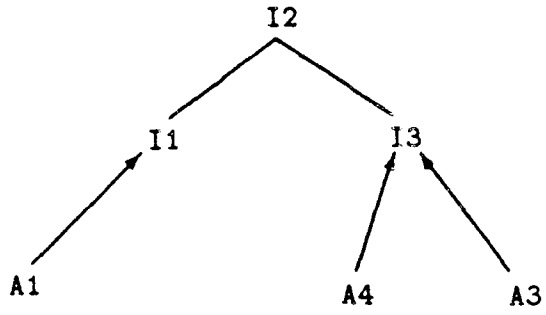


Figure 26: Inference Tree for Example Illustrating Multiple Derivations

Figure 26 shows the inference tree for this knowledge base. The relevant portion of the data structures of the MATMS at this point in the problem solving:

	C3	C4	C5
environment	(A3)	(A1 A3)	(A3 A4)
inferences	(I3)	(I1 I2 I3)	(I3)
incompatibies	()	()	()

	A3	A4	I1	I2	I3
contexts in	(C3 C4 C5)	(C5)	(C4)	(C4)	(C3 C4 C5)
influences	(I3)	(I3)	(I2)	()	(I2)
assumptions	*	*	((A1))	((A1 A3)(A1 A4))	((A3)(A4))
preconditions	*	*	((A1))	((I1 I3))	((A3)(A4))

In general, adding a justification to an existing inference could be far more expensive than simply replacing an assumption with an inference because search must occur in both directions, instead of just up the tree.

Problem Solver Proposes Retracting Justification of Inference:

When an agent proposes retracting a justification for an inference, it is asking to remove a certain list of beliefs from the inference's *immediate preconditions*. If the justification exists, the MATMS must perform a potentially long series of operations. The easiest steps are the earliest. First, the justification is removed from the inference's *immediate preconditions*. Next, the *influences* slot of each belief mentioned in the justification the problem solver wishes to remove is readjusted. More precisely,

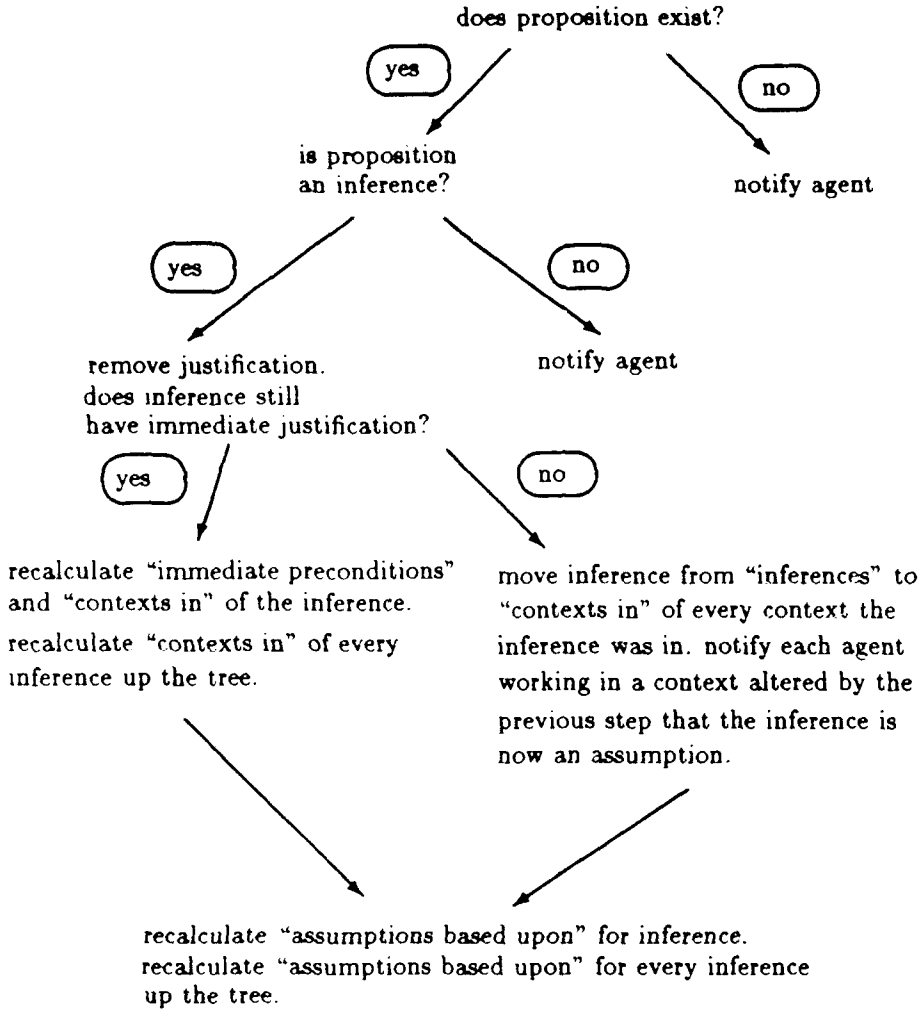


Figure 27: Decision Tree for *Problem Solver Proposes Retracting Justification*

the inference is removed from *influences* of each belief in the justification which is no longer mentioned in any member of *immediate preconditions* of the inference. From there, the steps become more costly.

If there is still at least one set of beliefs in the inference's *immediate preconditions*, then the *contexts in* of every inference up the tree (including the inference itself) must be recalculated, because it may no longer belong to a context currently in *contexts in*. An inference is removed from a context by removing the context from the inference's *contexts in* and removing the inference from the context's *inferences*.

If there is not a set of beliefs in the inference's *immediate preconditions*, then, conceptually, the procedure detailed earlier concerning replacing an assumption with an inference must be reversed. The only deviation from simple inversion of that algorithm is that for every context to which the inference previously belonged, the inference is replaced by the assumption. Also, if a problem solver is currently working in one of these contexts, it must be notified that an inference it was working with is now an assumption.

Next, every inference "up" the tree (including the inference from which the justification has been removed) must have its *assumptions based upon* recalculated, as well as a possible adjustment to *contexts in*.

4.5.2.5.4 Read Operations The discussion of the MATMS would not be complete unless its knowledge access functions were discussed. The read operations, from the perspective of the MATMS, are simple. The problem solver has the much more difficult task of deciding what to ask, and how to ask it.

The most important (and usually the most difficult) feature of accessing MATMS data is the domain dependent mapping from the problem solver data implied in the query to the relevant set of beliefs. This mapping results in both knowledge which has been explicitly stated by a problem solver in the course of normal problem solving and the default knowledge. A domain-specific mapping function which relates to a frame based knowledge base implementation will be discussed in detail in a subsequent section. Once the set of relevant beliefs has been determined, the rest is straightforward.

Read operations can be viewed as falling into one of three categories: is a particular problem solver datum contained in a particular problem solver's belief set? (problem solver dependent query); what problem solvers currently believe a particular problem solver datum? (all problem solver query); and describe all beliefs relevant to a particular problem solver datum (context independent query).

For a *problem solver dependent query*, after determining which beliefs are relevant to the problem solver data in question, the context of the problem solver to which the query refers is consulted. First, the non-default beliefs are considered. If any of these are contained in the context, the MATMS replies appropriately. The MATMS could respond with more than one belief, and the beliefs could be contradictory. If none of these beliefs are contained in the context, then the problem solver is considered "opinionless", and only the default knowledge is returned if it exists. If default knowledge is returned, it is identified as such in the response.

For the *all problem solver query*, a problem solver dependent query is performed for all problem solvers.

An *all context query* is used when an agent requires all relevant beliefs concerning

a particular problem solver datum. All beliefs, including default beliefs which are relevant, are returned. If the belief is an inference, then its derivation is returned. Assumptions are simply returned, identified as assumptions.

The problem with the responses to an *all context query* is that a problem solver may not understand many of the intermediate steps used to derive the inference. It may not even understand the assumptions the inference is based upon. Perhaps a more useful query is a variation, such as the problem solver simply giving the MATMS a set of beliefs, and then asking what a particular problem solver datum would be if the beliefs were "true". In other words, the problem solver might ask something of the form "Suppose A and B were true. What would be the value of C?" If C can be derived from A and B either directly or indirectly, the MATMS responds accordingly. If C is an assumption in the knowledge base, then the MATMS would respond that it is an assumption and its validity is thus not connected to A or B. If no logical connection between A, B, and C has been registered with the MATMS, then it would reply only the default value for C if it exists.

4.5.2.6 Using The MATMS The MATMS was designed to be used by agents that "understand" a specific set of operating constraints. For this reason, unless it is used properly, some features of the MATMS might be lost. This section discusses how the MATMS should be used by agents without constraining how problem solvers should be written. Problem solving can take place in a variety of forms, so a presentation of how exactly it should be done is impossible. Rather, this section details some basic aspects of problem solving and in particular what a problem solver should expect the MATMS to do and reply when the problem solver interacts with it. Whereas the previous section was written from the viewpoint of the MATMS, this section is from the viewpoint of a problem solver.

4.5.2.6.1 Designing an Appropriate Problem Solver A problem solver which would work well in the MATMS setting must be rational. In terms of read and write operations, it also must be aware of what to expect from the MATMS, and how to use the MATMS.

Rationality:

The problem solver can only expect rational results from the MATMS if its inference mechanisms are rational. That is, supplying the MATMS with inferences which contradict each other logically on the basis of the inference's *immediate preconditions* will cause the MATMS to act irrationally. This should be expected, since the MATMS is only reflecting what it is supplied with.

The general rule is that a problem solver cannot produce inconsistent inferences from the same set of assumptions. The concept of rationality can be illustrated by a number of examples.

1. $AB \Rightarrow C$
   $AB \Rightarrow D$
   $CD \Rightarrow \perp$

   A problem solver possessing these rules would be clearly irrational. Two rules acting on the same set of preconditions cannot result in contradictory expressions.

2. $AB \Rightarrow E$
   $CD \Rightarrow F$
   $EF \Rightarrow \perp$

   At the other end of the spectrum, a problem solver with these rules is rational. Two rules can act on completely different sets of beliefs and result in differing expressions. Comparisons of most problem solving rules fall into this category.

3. $AB \Rightarrow C$
   $AD \Rightarrow E$
   $CE \Rightarrow \perp$

   This example falls in between the extremes characterized by the first two examples. These inference rules are rational when compared to each other.

4. $AB \Rightarrow C$
   $ABE \Rightarrow D$
   $CD \Rightarrow \perp$

   While these rules also fall in between the extremes given by the first two examples, this set is irrational according to the manner in which the MATMS operates. The first rule states that in the presence of A and B, the MATMS should include C. The second rule states that in the presence of A, B, and E, include D. With these rules, the context of the environment ABD will include C and D, which is irrational.

Rationality as discussed here is conceptually not difficult to encode in a problem solver, for it requires only that the inference rules of a problem solver be rational as compared to each other. This property will be referred to as *self-rationality.*

Assumptions:

During normal problem solving activities, an agent can be expected to make or use assumptions. A problem solver makes assumptions when it is unsure of a particular piece of knowledge. There are three points which need to be made concerning assumptions.

First, consider the example situation. An agent has a particular rule:

$$ABCD \Rightarrow E$$

If the agent currently believes A, B, and C, then it could assume that E is valid, also. Even though an inference rule indirectly has produced E, E should not be registered as an inference. An inference is meant to represent that all preconditions of a rule have been met, which is not the case in this example.

This leads to the first point regarding assumptions. Problem solvers should internally record why a particular assumption was made. It is important to realize that the MATMS should be used to record the assumption itself, not the reasons why the assumption was made. A problem solver should simply record E with the MATMS, and internally maintain the knowledge that E was assumed because it has a rule ($ABCD \Rightarrow E$) which had most of the preconditions necessary for its firing believed.

Second, an agent should never attempt to add a default assumption to its belief set, because it is most likely already present. If it is not, because the agent has overridden the default, then the appropriate way of reasserting the default is by retracting the belief which overrides it.

Third, if the problem solver is presented with contradictory default assumptions, the manner in which it should register with the MATMS that it believes one default in particular is by accepting beliefs which directly override the defaults which the agent does not accept. This is a little awkward, but overall the best procedure. In general, inconsistent default assumptions should be avoided.

Interacting with the MATMS:

When a problem solver changes its assumption set, it should always register the change with the MATMS so that the MATMS can either remove or add inferences as necessary and determine if the new belief set is consistent. The MATMS will always confirm the transaction with the problem solver in one way or another. This confirmation might include a message indicating that the problem solver may have made a mistake, such as when the agent attempts to add an assumption to its belief set which is already present. If the resulting belief set is inconsistent, the MATMS will inform the problem solver that certain subsets of the belief set are inconsistent. Each belief in each subset will be described. Each description includes the assumptions from which the belief has been derived. It is up to the problem solver to determine which assumptions it wishes to remove in order to make its belief set consistent.

When an agent registers an inference with the MATMS, the inference rule used to generate the inference should not be included (this was proposed in [16] as a method to record the control sequences used to generate knowledge.) This would be counter-productive to the overall system, because an inference of one agent should not be inherited by another unless it explicitly included the other's inference rules in its belief set.

Determining which read operation to perform in a given situation is difficult. When an agent queries its own beliefs of itself as well as other problem solvers, the *problem solver dependent query* should be used. If a problem solver must assess the overall belief state of a particular piece of knowledge, the *all problem solver query* should be considered. To determine what any problem solver has ever believed concerning a particular piece of knowledge, the *all context query* should be used.

4.5.2.6.2 Designing a System around One MATMS Designing a problem solving system around one MATMS should be influenced by two topics: mutual rationality and inconsistency across problem solvers.

Mutual Rationality:

It has already been discussed that a problem solver must be self-rational. In addition, a problem solver must be rational as compared to the others for essentially the same reasons as were mentioned in the case of a single problem solver. This is true because inferences are not problem solver dependent. We say that two problem solvers are *mutually rational* if the inference rules of each problem solver could be used to create a self-rational problem solver.

When considering a single problem solver, requiring rationality is not unreasonable. If any problem solver in any system were not rational, it would probably not be very productive. However, the criteria that problem solvers be rational when compared to one another is a somewhat more restrictive and difficult to achieve. Two problem solvers could for instance be self-rational, but be irrational when compared to each other. To ensure mutual rationality, problem solvers must be designed in accordance with overall system goals; coordination of design is essential.

Resolving Inconsistency among Problem Solvers:

Mutual rationality suggests only that the inference rules of one problem solver be consistent with all other problem solving rules. For instance, given the same preconditions, two inference rules should not produce two pieces of knowledge which are contradictory. Mutual rationality includes nothing about what assumptions each problem solver can choose with which to work. This allows each problem solver to

perform in a variety of problem solving activities, fairly independent of the activity of the others.

Two problem solvers which are mutually rational could seemingly present contradictory beliefs to the MATMS. For instance, suppose one problem solver had presented "it's 90° out" as an assumption in its current belief set, and had then inferred "it's a good day to go swimming because it's 90° out". Suppose that another problem solver had told the MATMS that its current context belief set includes "it's 40° out", and had then inferred "it's a bad day to go swimming out because it's 40° out". Clearly each problem solver is self rational, as well as mutually rational when compared to the other. In addition, the MATMS would "support" the context of each problem solver, because the belief set of each problem solver is self-consistent.

However, there is clearly a problem with the overall problem solving. The individual problem solving is diverging if indeed the first problem solver believes that it is truly 90° outside, and the second believes that it is 40° (neither agent is involved in hypothetical reasoning). Certainly the MATMS recognizes that there are two assumptions in the knowledge base, 90° out and 40° out, which can not exist in the same context because they are directly contradictory. The conflicting beliefs are not present in the same context, so there is no problem from the viewpoint of the MATMS.

Any time the MATMS is used, a single problem solver should monitor the true world assessments of the others and recognize when inconsistencies between problem solvers arise. This problem solver is essentially part of the domain independent MATMS, except when considering that the rules necessary to resolve the conflicts must be domain dependent.

Two architectures could be investigated, depending upon the domain. These architectures are shown in Figure 28, with the agent responsible for resolving inconsistency labeled as PSC. The architecures differ primarily in the interagent communication paths utilized. In the first, each agent interacts directly with the MATMS to get its beliefs. This would be faster for the individual problem solvers, but would also make the job of the agent resolving the inconsistencies difficult. In the second, each problem solver interacts through the agent resolving the inconsistencies to communicate with the MATMS. Monitoring beliefs is thus much easier.

The agent responsible for maintaining consistency across local problem solvers can certainly recognize inconsistencies between problem solvers. To resolve them, either of two methods could be used. The PSC could itself choose one value over the other without consulting the two problems solvers from which the beliefs originated. Alternatively, the PSC could tell the problem solvers that they conflict with one another, leaving resolution up to the inconsistent problem solvers. Both methods require further research.

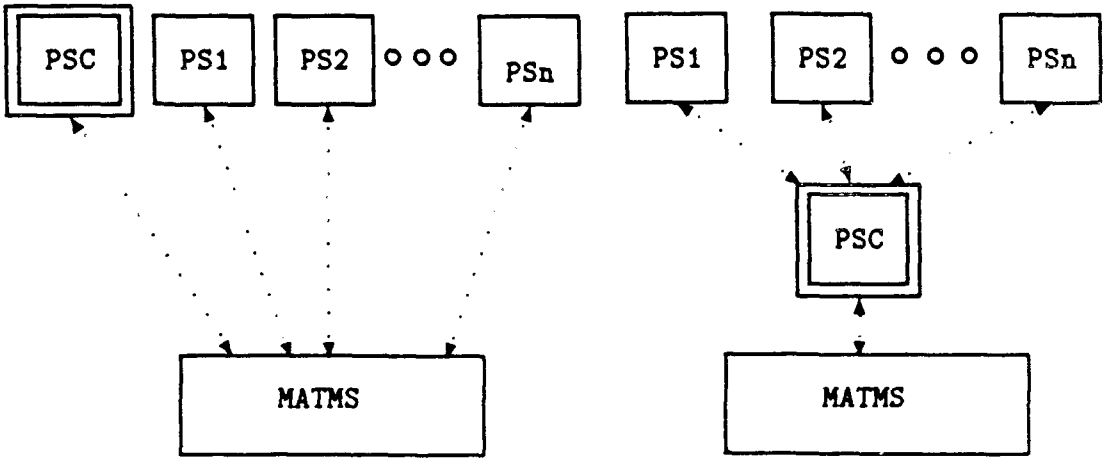


Figure 28: Proposed Interagent Communications Paths

4.5.2.7 Example Implementation

4.5.2.7.1 The Domain The domain for which the MATMS has been implemented is a distributed knowledge based system for managing a large-scale communications network. The communications network provides telecommunications service for people as well as machines. The communications system can be described on three levels.

On one level, the communications network can be viewed as a sparsely interconnected array of transmission facilities called *sites*. Each site is generally only connected to one or two other sites. The interconnections between sites (*links*) provide a transmission medium over which to send communications signals between sites. For control purposes, sites are grouped into non-overlapping sets called *subregions*. Each subregion contains one SubRegion Control Facility (*SRCF*) to which each site in the subregion reports the operational status of its equipment, availability of resources, etc. A portion of a "typical" communications network is presented in Figure 29.

Another level is the equipment configuration at each site. This level includes the "barebones" equipment and connections necessary to originate and switch communications signals. An example equipment configuration is given in Figure 30 (adapted from [10]). Equipment and their interconnections are constrained by a variety of rules.

The third level forms the communications path level which is probably the most important, for it can be considered the fundamental view of the network. The two primary objects present at the communications path level are *trunks* and *circuits*.

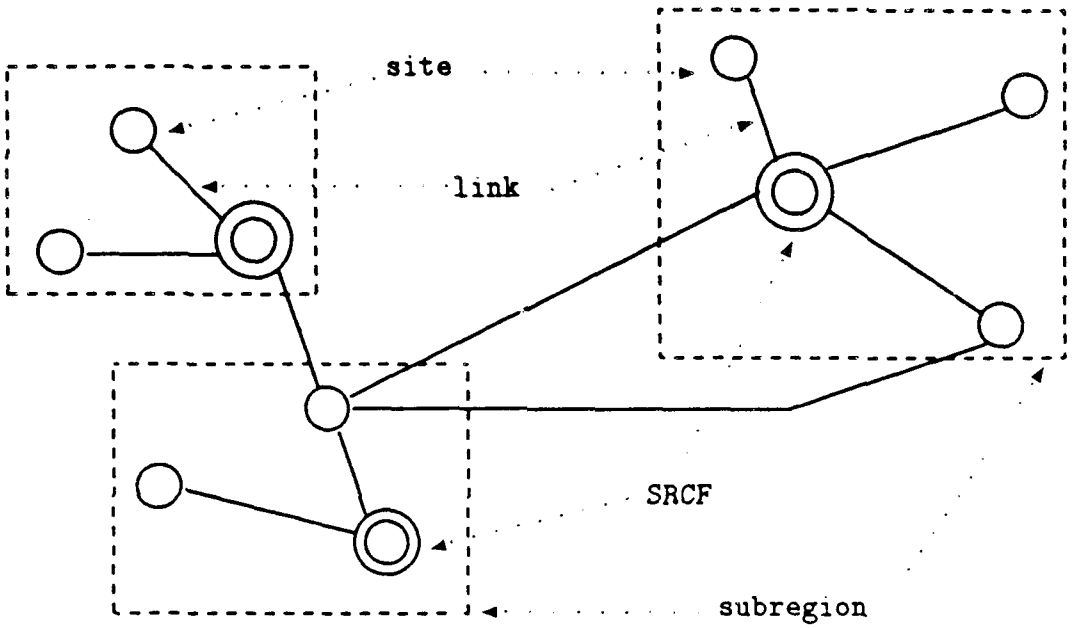


Figure 29: Sample Communications Network

A circuit is the complete elementary path between two pieces of terminal equipment by which two-way telecommunications service is provided. A trunk is a group of equipment and connections which establishes telecommunications connectivity by providing a resource for circuits to ride. Circuits ride on channels of a trunk; there is typically a capacity for several channels per trunk. A useful analogy for channels on a trunk is to imagine one big pipe (the trunk), which contains a number of small pipes (channels) running the entire length of the big pipe. With this analogy, it is easy to see that a trunk can ride channels of other trunks. The trunk exists with or without the circuit, but this is not symmetric; a circuit cannot exist unless it rides a trunk, or a list of trunks connected in series. The trunk refers to "physical" connectivity, whereas the circuit refers to "logical" connectivity. For more details, see [38].

In order to understand how the MATMS operates in this domain, one should understand the general concepts of the communications path level. Thus, an example of two sites partially configured is presented in Figure 31 with careful attention to trunks and circuits. The description of Figure 31:

- *ckt1* (connecting *user1* and *user2*) rides *trkx* which rides *trky* which rides *trkz*.
- *trkx* starts at *m981* and ends at *m982*. It has 24 channels.
- *trky* starts at *m991* and ends at *m992*. It has 8 channels.

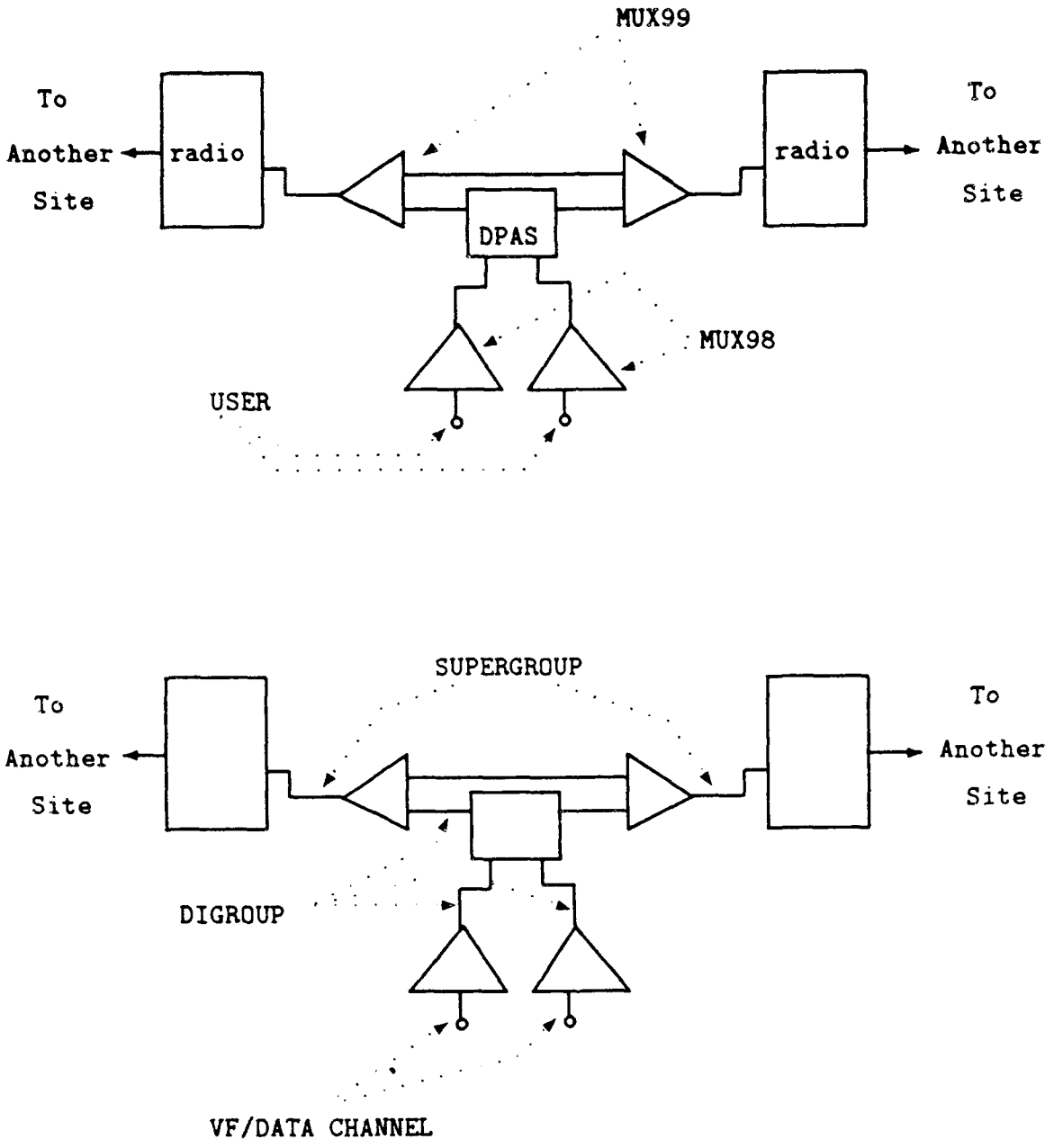


Figure 30: Sample Equipment Configuration

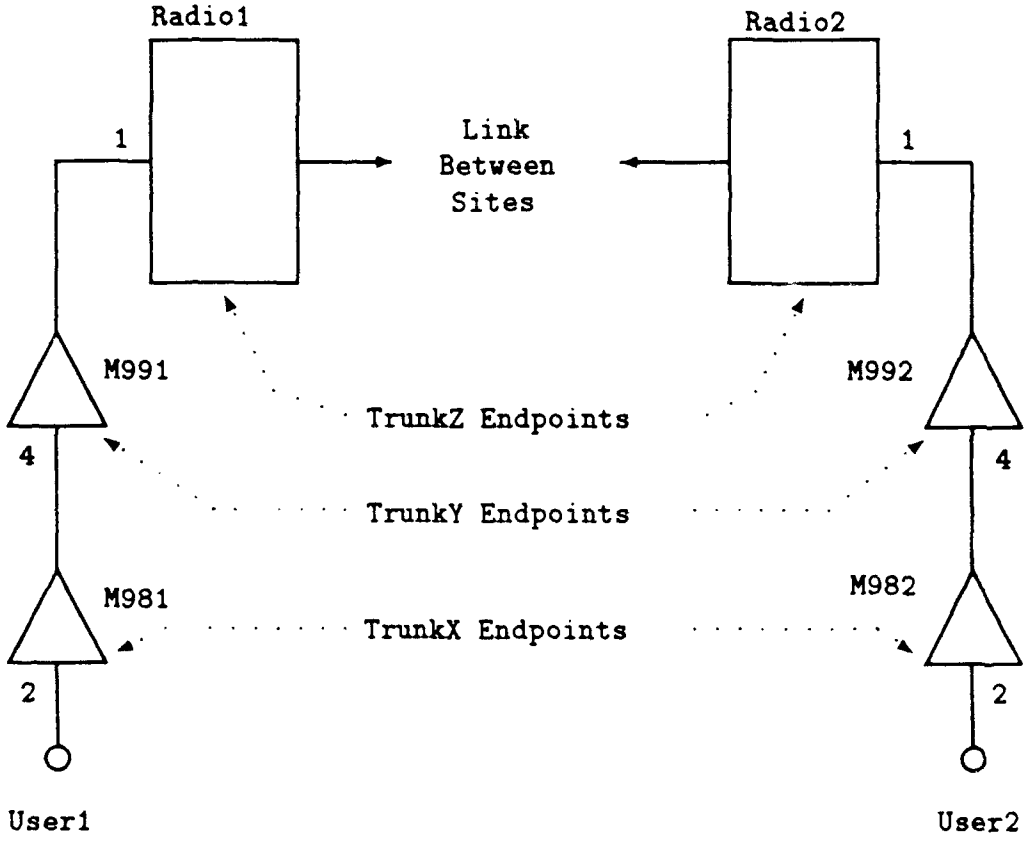


Figure 31: Sample Trunk and Circuit Configuration

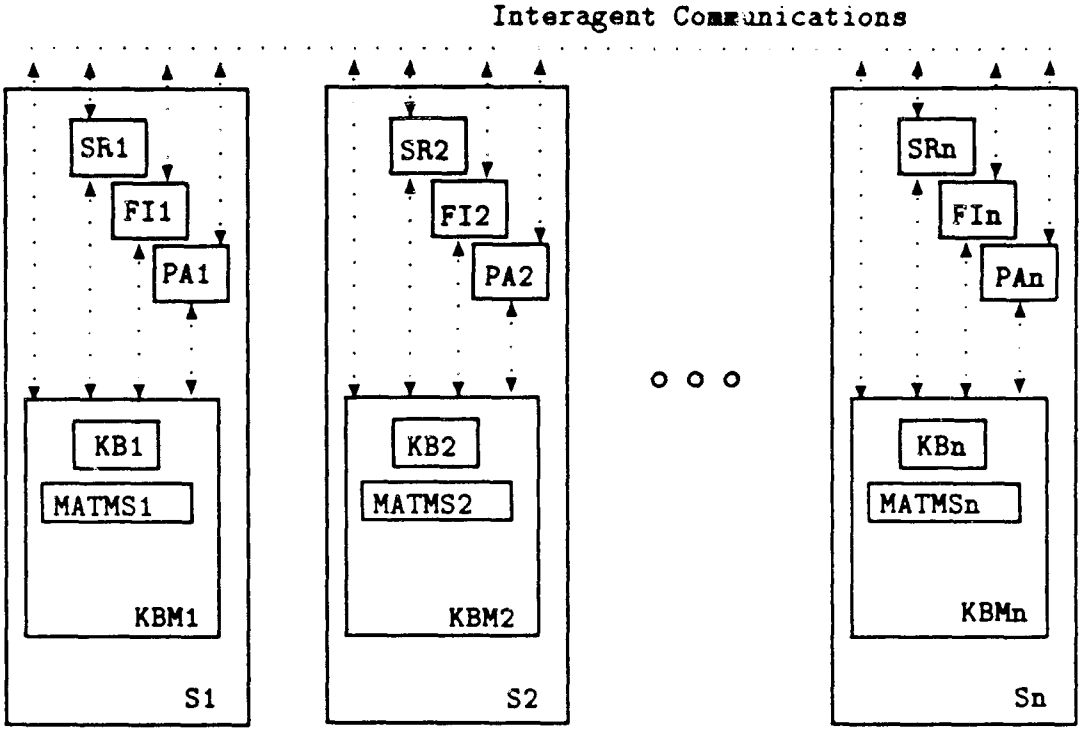


Figure 32: Communications Network Knowledge-Based System Architecture

- *trkz* starts at *rad1* and ends at *rad2*. It has 2 channels.

If any piece of equipment should fail or degrade to the point of causing any of *trkx*, *trky*, or *trkz* to fail, then the two end users of *ckt1* would be disconnected. The circuit would be said to be *disrupted*.

Maintaining communications service is the primary objective of the system. If any circuit fails, then the circuit must be restored often at the same time as the problem is being diagnosed. Restoration could proceed by reconfiguring equipment, or selecting existing alternative trunks, or a combination of both.

4.5.2.7.2 Knowledge-Based System Architecture The knowledge base system architecture was briefly discussed in Section 4.3.1 The purpose of this section is to expand on that description. In particular, the role of the KBM will be discussed further.

The architecture of the knowledge-based system is shown in Figure 32. The inter-

agent communications paths have been included in the figure to convey the general operation of the system. Although agents can communicate with other agents of the same type at different subregions, most communication is with the local Knowledge Base Manager (KBM).

The KBM has many responsibilities. It grants access to the knowledge in the local knowledge base, must process the transactions from the SR, PA, and FI agents in a logical order, must know where knowledge requested by the agents resides (if it not present within the local knowledge base), and most importantly it has to maintain a consistent local view of the state of the world by monitoring the beliefs of the individual agents in the node in combination with other KBMs in the network.

To aid in maintaining a consistent local view, the KBM includes the MATMS. The MATMS is used to keep the belief sets of each of the agents consistent, as well as to recognize inconsistency when comparing belief sets. It is the KBM, however, which must attempt to recognize and resolve the inconsistency. The role of the KBM after recognizing discrepancies is to advise the problem solvers as to how to resolve the inconsistencies, often by consulting other KBMs where appropriate. For example, if PA believes a certain trunk is not operable, and FI believes it is, the KBM might ask another KBM what it believes the state of the trunk to be. This of course would only work if the trunk crossed subregion boundaries (so that another KBM *would* have knowledge of it), and the KBM knew which other KBM to ask.

4.5.2.7.3 Architecture Implementation The global knowledge base is created using the Graphical User Interface for Structural Knowledge (GUS [24]) on a Symbolics 3670. An option in GUS divides the knowledge base along subregion boundaries in order to create the knowledge bases for each problem solving system discussed in the previous section. Division of knowledge can be modified to test different distributed knowledge representation schemes. The knowledge representation scheme in GUS, which is frames, is utilized in the distributed knowledge bases.

A distributed simulation environment (SIMULACT [27]) is used to test the knowledge based system. It provides a parallel simulation environment for any number of agents. Interagent communications support is provided in a highly flexible format.

The Knowledge Base Manager (KBM) comprises a single agent in SIMULACT. At this time, it contains query processing functions, the knowledge base itself, and the MATMS. Knowledge provided to the KBM from GUS is treated partially as constants and partially as default assumptions to the MATMS. For example, a particular radio's name is constant, as is its operational definition, while its initial status is treated as a default assumption. In the absence of information to the contrary, the status of a radio is assumed to be the value provided by GUS.

The MATMS exploits the knowledge representation scheme (frames) and to some

extent the domain itself. In particular, the task of finding all the beliefs which are relevant to a particular problem solver query is handled by realizing that most queries will access a particular slot in a frame. Therefore, beliefs which are relevant to a given slot are kept within the slot along with the default. This will be illustrated in the section which follows.

4.5.2.7.4 Examples of MATMS Usefulness

Example 1:

The purpose of this example is to illustrate the way data structures are accessed.

Suppose that there is a frame for a particular instance of a trunk:

id:	trk9
is-a:	trunk
type:	digroup
.	.
.	.
.	.
status:	≫ MATMS default frame≪

The default frame is:

default:	UP
beliefs:	()

The default frame can be interpreted as "Trk9 is up by default. There are no beliefs presented by problem solvers to override the default."

Now suppose that PA asks the KBM for its (PA's) belief concerning the status of trk9. The KBM would invoke the MATMS by attempting to access a slot of a frame which is controlled by the MATMS. In other words, the KBM would attempt to access the slot, which automatically invokes the MATMS. Because there are no beliefs in the context which PA is currently working in (this must be the case because the *beliefs* slot of the default frame is empty), the response would be "UP, by default."

If PA subsequently proposed the assumption "the status of trk9 is down", the MATMS would change the status frame to be:

default:	UP
beliefs:	(*A1*)

where *A1* corresponds to "the status of trk9 is down."

If PA asked again what the status of trk9 is, the KBM would reply "down" (the KBM would invoke the MATMS to determine the status of trk9, which would find *A1* present in the environment of the current context of PA). However, if FI asked, it would still get "UP, by default." Note that if for some reason FI proposed that the trunk is down, the status slot of trk9 would remain the same. FI would just have *A1* added to its current belief set. Incidentally, if these were the only transactions which the MATMS had made, then the end result is that FI would be placed in the same context as PA.

Example 2:

Example 2 shows more of the potential of the system and is tied more closely to the domain.

Suppose that the following configuration exists for a subregion (Figure 33). A general description is that *ckt*1 rides *trkz*, *trkw*, and *trkx*. *Ckt*2 rides *trky*, *trkw*, and *trkx*. *Ckt*3 rides *trkx*, and then *trkv* to another subregion.

Initially, FI, SR, and PA have not made any assumptions about the communications network. Each problem solver is working in C1 (context 1). Each time a new context is created by the MATMS, the context counter is increased by one. For instance, the next context created will be named C2, and so on. The knowledge base consists of the defaults (though only the defaults relevant to the discussion are mentioned here):

trkw is up	[A1]
trkv is up	[A2]
trkz is up	[A3]
trky is up	[A4]
trkx is up	[A5]
*ckt*1 is up	[A6]
*ckt*2 is up	[A7]
*ckt*3 is up	[A8]

In addition, the following premises have been entered into the MATMS: a trunk cannot be up and down (P1), a circuit cannot be up and down (P2).

Now, at time t_0, suppose PA is notified of a user alarm concerning *ckt*1, at the same time that it is notified of a user alarm concerning *ckt*2. (A *user alarm* is when the user of a circuit notifies a technical control facility to complain that he has lost service in a particular circuit.) PA views user alarms as assumptions in the form of new observations of the world, and immediately registers the assumptions with the MATMS.

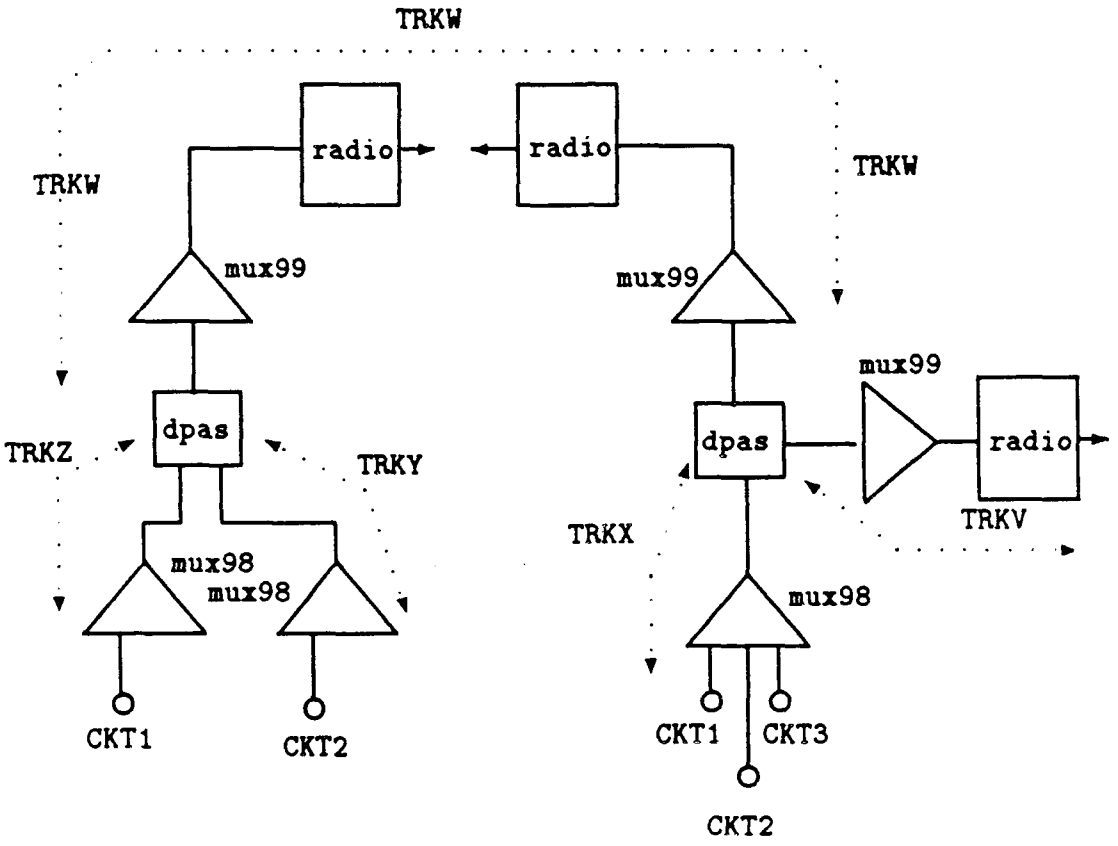


Figure 33: Communications Network for Example 2

user alarm *ckt*1 at t_0	[A9]
user alarm *ckt*2 at t_0	[A10]

Adding the two assumptions [A9] and [A10] results in contexts C2 and C3, respectively.

PA continues working and eventually registers the following inferences with the MATMS, through the KBM. Even though there was not a user alarm, PA concluded that *ckt*3 was down through the application of the rule "If multiple circuits on a trunk fail at the same time, assume the trunk has failed." Therefore, PA asserts that *trkw* and *trkx* are down. Asserting that *trkx* is down leads PA to infer that *ckt*3 is also down.

*ckt*1 is down because	user alarm *ckt*1 at t_0	[I1]
*ckt*2 is down because	user alarm *ckt*2 at t_0	[I2]
*ckt*1, *ckt*2 fail at same time because	user alarm *ckt*1 at t_0 user alarm *ckt*2 at t_0	[I3]
trkx is down		[A11]
*ckt*3 is down because	*trkx* is down	[I4]
trkw is down		[A12]

Note that *ckt*1 *is down* does not represent a contradiction of beliefs for PA in the MATMS, because *ckt*1 *is up* was a default assumption.

The end result of this is that PA is placed in a context which is defined by:

name:	C5
environment:	(A9 A10 A11 A12)
inferences:	(I1 I2 I3 I4)
incompatible belief sets:	()

The knowledge base is briefly described as:

	A9	A10	A11	A12
contexts in	(C2 C3 C4 C5)	(C3 C4 C5)	(C4 C5)	(C5)
influences	(I1 I3)	(I2 I3)	(I4)	()

	I1	I2	I3	I4
contexts in	(C2 C3 C4 C5)	(C3 C4 C5)	(C3 C4 C5)	(C4 C5)
influences	()	()	()	()
assumptions	((A9))	((A10))	((A9 A10))	((A11))
preconditions	((A9))	((A10))	((A9 A10))	((A11))

Because it has no reason to disbelieve PA at this time, the KBM accepts the assessment of the current state of the world by PA as correct. In other words, it adopts the belief set of PA by asking the MATMS to place it in the same context as PA.

When PA is finished assessing the user alarms on the circuit operation, it tells FI to begin work. FI begins by asking the KBM for its current assessment, which happens to be solely PA's assessment. Therefore, FI is placed in context C5. Thus, it inherits the beliefs which assert that *ckt*1 is down, *ckt*2 is down, *trunkx* is down, etc.

FI performs measurements on each of the circuits or trunks in question and determines that *ckt*3 is actually up, not down. FI enters the following beliefs into the knowledge base:

tests of *ckt*1 at t_1		[A12]
tests of *ckt*2 at t_1		[A13]
tests of *trkx* at t_1		[A14]
*ckt*1 is down because	tests of *ckt*1 at t_1	[I6]
*ckt*2 is down because	tests of *ckt*2 at t_1	[I7]
trkx is up because	tests of *trkx* at t_1	[I8]

A more careful, step by step analysis of the steps involved when FI changes its belief set is necessary to understand the operations of the MATMS.

1. FI adds A12 to its belief set, which causes C6 to be created. The KBM accepts A12 into its belief set.

2. FI adds A13 to its belief set, which causes C7 to be created. The KBM accepts A13 into its belief set.

3. FI adds A14 to its belief set, which causes C8 to be created. The KBM accepts A14 into its belief set.

4. I6 adds another justification for "*ckt*1 is down". It serves as confirming evidence, as does adding I7. Because the KBM has accepted the beliefs upon which both I6 and I7 are based upon, it automatically inherits I6 and I7.

5. When FI attempts to add I8, the MATMS responds that its belief set is inconsistent, because it currently believes that *trkx* is up, and *trkx* is down. The MATMS marks the current context of FI (C8) inconsistent. The MATMS informs FI that it can remove "*trkx* is down" directly, because it is an assumption, and it can remove "*trkx* is up" by retracting "tests of *trkx* at t_1". For FI, there

is no great difficulty in deciding that "*trkx* is down" should be removed from its belief set, because "*trkx* is up" is an inference which it just made.

However, the KBM is faced with maintaining consistency. At this point, FI believes that "*trkx* is up", and PA believes that "*trkx* is down". In this situation, the KBM clearly believes FI, because PA is prone to errors due to time constraints. That is, PA is forced to make a fast, rough estimate, while FI must be certain of its work before it enters beliefs into the knowledge base. For that reason, the KBM follows FI and retracts "*trkx* is down". Note that KBM no longer believes that "*ckt3* is down". Further work by FI would suggest that *trkv* is operating properly.

At this point, FI tells SR that it is finished. SR begins work to restore circuits *ckt1* and *ckt2*.

4.5.2.8 Status There are limitations to the current design of the MATMS which should be investigated further. Specifically, premises need to be extended, and overall system efficiency should be investigated further.

The current premise structure allows for two pieces (or classes) of problem solver data to be considered inconsistent. For instance, "a trunk cannot be up (operational) and down (inoperable) at the same time." This causes difficulties when one attempts to model a more complex premise structure which involves more than two objects. A one-OBof-three situation (choosing one route from three choices) or perhaps a two-of-three situation (having enough money to buy any two of a fishing pole, a softball glove, and a tennis racket, but not all three) cannot be handled currently. This has not been attempted yet because there is no immediate need for it; the premises necessary for our application demand a simple binary situation.

Although the design clearly addresses the importance of an efficient system, the degree of efficiency is still questionable. It is not obvious whether the inefficiencies present are a result of the implementation or the design. It is clear that the system is often forced to recompute justifications for inferences. This could involve a large amount of time in order to keep the inference trees as compact as possible.

There was always some debate about including default assumptions explicitly in the environment of contexts. If they were included, creating contexts could be performed more quickly than in the present design, but there would be more pointers between objects in the knowledge base. On the other hand, not including default assumptions explicitly in contexts increases the ease of comparing belief sets and in general makes the operations of the MATMS more efficient. If, upon testing the system more, it is determined that the MATMS spends most of its time creating contexts, an argument could be made for a redesign which would put default assumptions explicitly in contexts. This modification would not be difficult to make.

Many of the questions involving the efficiency of MATMS have not been addressed because the problem solvers (FI, PA, SR) have been under development at the same time. Interactions involving the MATMS have only been tested with the SR problem solver, so such concerns as multiple derivations, context switching, and to some extent multiple context maintenance have only been tested using "typical" cases which have been fabricated. As the knowledge based system matures, the MATMS will surely be refined, although the design appears to be sufficiently flexible to handle most changes and was designed to be adaptable.

4.5.3 Distributed Automated Reasoning System (DARES)

Issues related to distributed problem solving in geographically and/or logically distributed domains are addressed with respect to the role of knowledge. As a vehicle to explore these issues, a Distributed Automated REasoning System (DARES) has been implemented. This system can be viewed as a collection of distributed agents which interact to perform automated reasoning about the distributed domain. DARES' architecture and its knowledge acquisition heuristic are discussed. A detailed example of how DARES performs automated reasoning in a distributed domain is given, and preliminary results based on DARES' performance are also presented.

The purpose of this work is to explore issues related to the role of knowledge in distributed problem solving domains which are geographically and/or logically distributed. The problem solving environment in such a domain can be viewed as a collection of semi-autonomous, loosely coupled, intelligent agents. Each agent has its own partial view of the domain, and no one agent has complete knowledge. These agents use a message passing paradigm to cooperate and coordinate their problem solving activities towards the satisfaction of one or more concurrent goals or tasks. In domains of this kind, there is an underlying assumption that no single agent can achieve the task at hand by itself. Cooperation is required for effective problem solving. This work is concerned with assessing the degree to which shared knowledge impacts problem solving behavior and the degree of sensitivity a distributed problem solving system has to where nonlocal knowledge resides.

In this research we use a Distributed Automated REasoning System (DARES) as a vehicle for investigating the role of knowledge in distributed problem solving. This system can be viewed as a collection of distributed agents which interact to perform automated reasoning about the domain. Many concurrent reasoning tasks proceed simultaneously, and an agent may be involved in solving any subset of the complete set of tasks. Since the domain is distributed, there is no global view. Each agent has its own understanding or assessment of the domain expressed in the first order predicate calculus.

A distributed deduction system such as DARES was chosen to investigate the role

of knowledge in distributed problem solving, because many domains can be expressed using the first order predicate calculus. Furthermore, automated reasoning using the predicate calculus is a mechanical inferencing process which separates domain characteristics from reasoning strategies. Our goal is one of gaining insight into the role of knowledge in distributed problem solving which is domain independent.

There is a distinction between this work and other work on parallel theorem proving. Although speed considerations are important, they are not the primary criterion of performance in this work. Issues associated with qualitative information exchange and its impact on problem solving behavior are. Other work is being done in developing algorithms and architectures to perform theorem proving in parallel, which reduces the exponential nature of classical theorem proving to near linear characteristics [2, 6].

Our preliminary results indicate that distributed theorem proving is feasible and can be made complete. We have also observed performance characteristics that exceed those of a single agent performing a proof by itself with respect to number of resolvents generated and also with respect to the amount of time required.

These results are based on data being collected from our Distributed Automated Reasoning System referred to as DARES. In DARES, each agent has multiple instances of a saturation level, binary resolution theorem prover [41, 5], which uses a tautology and subsumption deletion strategy. DARES runs on top of SIMULACT [26], described in Section 4.4.1, which is a distributed testbed for the development of distributed systems.

We begin our discussion of distributed automated reasoning by first presenting a brief overview of traditional theorem proving and the resolution principle. A simple single agent example is given to illustrate these concepts. We then present the Distributed Automated Reasoning System's architecture, followed by an introduction to its distributed theorem proving strategies. The single agent example is then distributed over several agents and a detailed solution demonstrating how DARES performs distributed theorem proving is given. In conclusion we present our initial findings with respect to automated distributed deduction.

4.5.3.1 Automated Reasoning in Single Agent Domains Traditional theorem proving using the resolution principle was developed in the mid-1960s [41] and is based on the first order predicate calculus. Domain knowledge is represented by a conjunction of logical statements or *clauses*, where each clause is a disjunction of *literals*. A literal is an *atomic formula* or a negated atomic formula. Resolution is a rule of inference that is used to deduce a new fact from known information. For instance, if we know that C_1 and C_2 are TRUE, where $C_1 = (A \vee B)$ and $C_2 = (\neg B \vee C)$, then we can deduce that C_3 must be true, where $C_3 = (A \vee C)$.

We can deduce C_3 from C_1 and C_2, because C_1 and C_2 each contain the same literal, B, in complement form. If B is FALSE, then A must be TRUE according to clause C_1. Likewise, if B is TRUE, then C must be TRUE according to C_2. Since we do not know whether B is TRUE or FALSE, we don't know if A or C is TRUE. But we do know independent of B's value that $(A \vee C)$ will be TRUE for the reasons just discussed. This reasoning process is formally known as resolution.

Theorem proving uses resolution to determine if a given theorem can be deduced (logically follows) from what is currently known. It does so by a *refutation* process which assumes what it is trying to deduce is FALSE. It then uses the resolution principle to determine if there is a contradiction between this assumption and what it knows to be TRUE. If a contradiction is found, then the assumption must be incorrect, thus proving the theorem to be TRUE. A contradiction is detected when two single literal clauses resolve together producing the empty (or NIL) clause.

During theorem proving it is possible for the two clauses being resolved together to have literals which are comprised of variables. In order to perform resolution, these clauses need to have one literal, each of which is exactly the complement of the other. The process of binding these variables to values in order to make resolution possible is known as *unification*.

As an example of traditional theorem proving, as well as to demonstrate the associated concepts mentioned above, we now consider the following example of theorem proving in a single agent domain. This example [5] includes the following six axioms and negated theorem:

S1:	$\neg R(x) \vee S(x, f(x)) \vee T(x)$	(axiom)
S2:	$Q(f(x)) \vee \neg R(x) \vee T(x)$	(axiom)
S3:	$P(a)$	(axiom)
S4:	$R(a)$	(axiom)
S5:	$P(y) \vee \neg S(a, y)$	(axiom)
S6:	$\neg P(x) \vee \neg T(x)$	(axiom)
S7:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)

Without using a subsumption [5] and tautology [12] deletion strategy, this proof generates 825 resolvents before generating the NIL resolvent. When using subsumption and tautology reduction the solution space is pruned to 68 resolvents, of which only 6 are necessary in establishing the validity of the theorem. The resolvent tree structure is shown in Figure 34, and Figure 35 is a listing of all clauses appearing in Figure 34.

[5] This example is a modification to Example 5.22 on page 89 in [5].

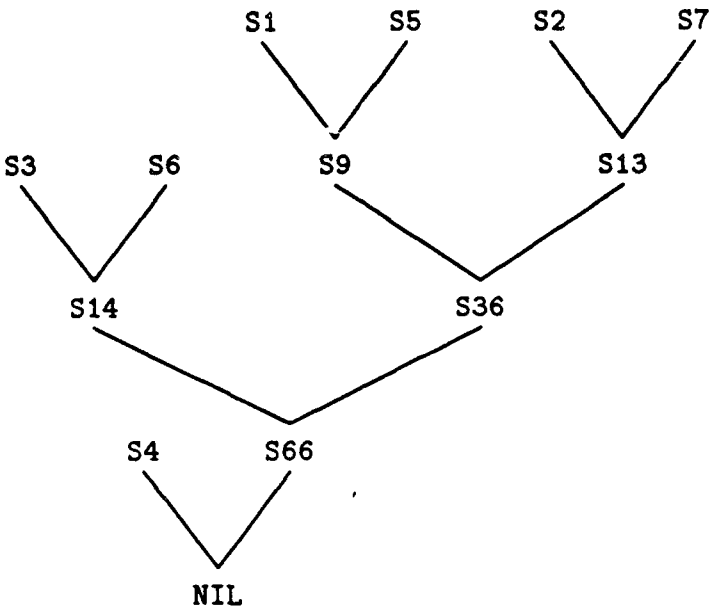


Figure 34: Proof of Example Theorem in a Single Agent Domain

Axioms, Negated Theorem, and Pertinent Resolvents:

S9:	$P(f(a)) \vee \neg R(a) \vee T(a)$	(resolvent of S1 and S5)
S13:	$\neg P(f(x1)) \vee \neg R(x1) \vee T(x1)$	(resolvent of S2 and S7)
S14:	$\neg T(a)$	(resolvent of S3 and S6)
S36:	$\neg R(a) \vee T(a)$	(resolvent of S9 and S13)
S66:	$\neg R(a)$	(resolvent of S14 and S36)
S75:	NIL	(resolvent of S4 and S66)

Figure 35: Solution Space Clause List for Single-Agent Example

4.5.3.2 Distributed Theorem Prover Architecture In a loosely coupled distributed system, an agent spends most of its CPU time in computation as opposed to communication. Since theorem proving by nature is computationally intensive, we have chosen a loosely coupled implementation for our distributed theorem prover. Each theorem prover agent spends the majority of its time performing binary resolution, with the balance spent on problem assessment and communication. Problem assessment helps determine what course of action to take next to further the proof locally, as well as to selectively transmit newly derived information of potential use to other agents. Determination and selection of this tendered information is based on previous requests made by an agent. Communication between agents generally falls into one of the following categories: (i) A request is sent to one or more agents for information; (ii) An agent returns information in response to a request; (iii) an agent offers new information based on past communications.

The architecture for the theorem proving agent has been tailored to suit the characteristics of the problem solving domain mentioned above (e.g. loosely coupled, multiple concurrent tasks). In Figure 36 the theorem proving agent is shown as being comprised of several processes attached to a communications network and a *REQUEST-QUEUE*. There is one mail process, with the remaining processes each being associated with a distinct problem solving activity. Each process has equal priority and active processes compete for CPU time in a round robin fashion. Under normal circumstances, every theorem prover process is active. The mail process is typically in a wait state and becomes active when new mail is received via the communications channel. Each theorem prover process has its own environment and is associated with one automated reasoning task identified by a unique tag. Theorem prover processes working on the same reasoning task throughout the network have the same tag. Furthermore, no two theorem prover processes for a given agent may work on the same theorem. It need not be the case that every agent works on every theorem. Figure 37 represents a fragment of a typical theorem proving network of agents connected only by a communications link (There is no shared memory among agents).

This architecture has proven to be extremely effective. The separate mail process eliminates the need for a theorem prover process to periodically check for incoming mail, which in itself is a complicated issue. By having a dedicated mail process, the processing of knowledge requests made by other agents is expedited, since attention to these requests is given immediately upon their arrival. We have also observed an increase in system performance when the CPU time to process incoming requests is minimized. It will be evident in our algorithms presented later that a theorem prover process making a request does expend effort in formulating the request, such that the CPU time required to process the request is minimized.

It can be seen in Figure 36 that all incoming mail is directed to the mail process.

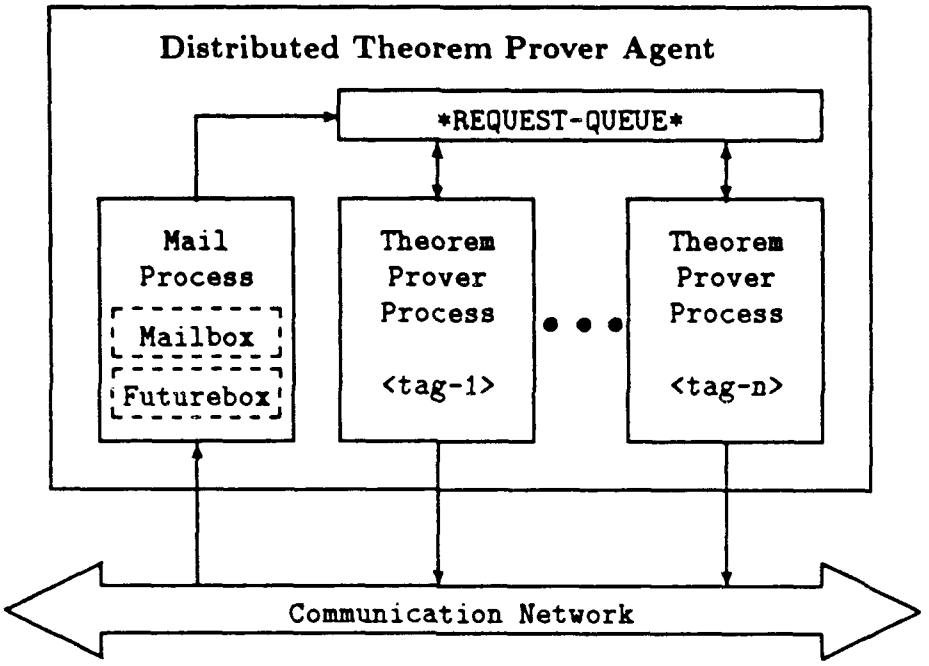


Figure 36: Architecture of a Distributed Theorem Prover Agent

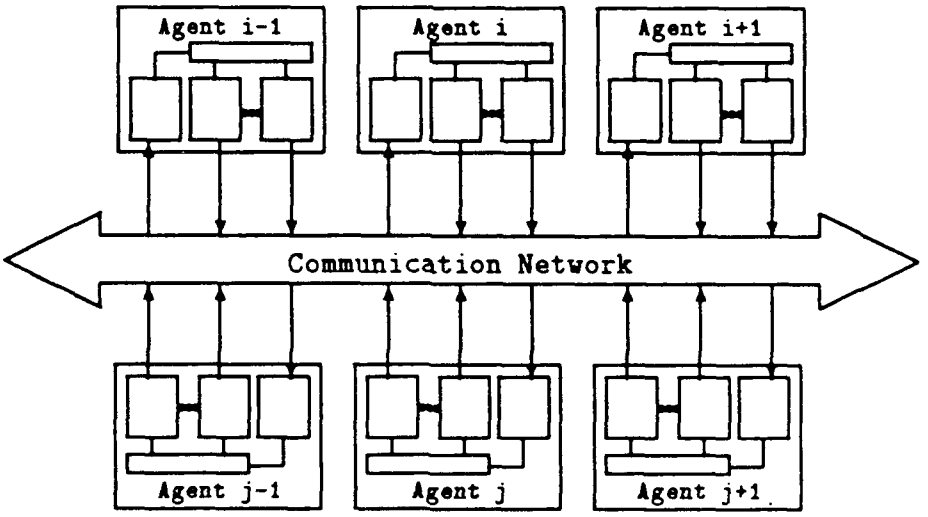


Figure 37: A Typical Network of Distributed Theorem Provers

Mail which is a one-way exchange between two agents is done using *memos*. A memo is an interaction between agents that does not require a reply. *Futures* [22, 42] are used to handle requests that do warrant a reply. A future is a data object that the requesting agent instantiates and retains on making a request. The reply is then routed back to that future when it is available. A future is tested periodically by the sending agent to ascertain when it is possible to extract the reply. *Future streams* are used in a similar manner as futures, but in situations where more than one reply is expected.

Theorem prover processes communicate directly with other agents via the communications network. Within an agent, theorem provers post requests to other theorem provers through the *REQUEST-QUEUE*. Entries into this queue are posted using theorem tags.

4.5.3.3 Distributed Theorem Proving Strategies As is the case with single agent theorem provers, theorem proving in distributed domains exhibits exponential behavior. It turns out, however, that some of the strategies used in classical theorem proving to help minimize the number of resolvents generated can also be used in the distributed case to reduce the content of information exchanged between agents. Development of these strategies are essential, since the performance of distributed theorem proving can be greatly enhanced by them. For instance, if the computational effort in replying to a request is significant, it may have not been worth making the request in the first place. Similarly, in information intensive domains such as distributed theorem proving, requests that receive a bombardment of information can be counterproductive.

In this section we outline our approach to distributed theorem proving. We will also introduce some distributed theorem proving concepts that are directly related to well-established classical theorem proving reduction strategies. Since this is a general introduction to distributed theorem proving, a detailed example will be given in the next section.

Figure 38 is a flow diagram which depicts a high level view of our approach to distributed theorem proving. Essentially, the top left loop in the diagram is no more than a flowchart for the traditional saturation level type theorem prover. In traditional theorem proving, resolution is done one level at a time. When a NIL resolvent is generated, the system halts with the theorem having been established. When there is no NIL resolvent and the current level of resolution has exhausted all possibilities, the newly generated resolvents are then used to begin the next level of resolution. This process continues until either a NIL resolvent is eventually obtained, or the current saturation level fails to generate any new resolvents.

We have previously stated that in our distributed environment no one agent can achieve the task at hand by itself. Therefore, we do not terminate the resolution

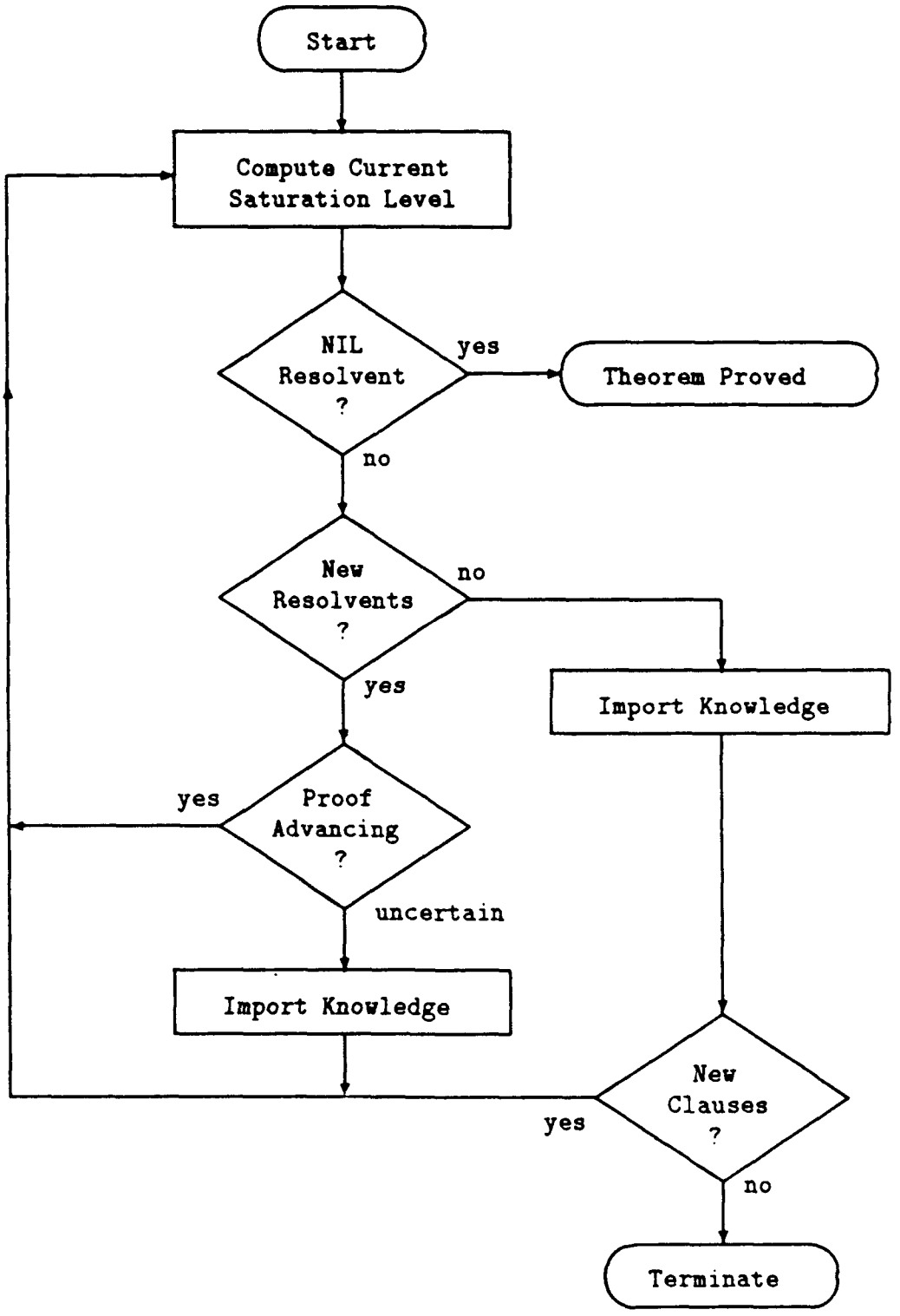


Figure 38: Distributed Theorem Proving Flow Diagram

process simply because the current resolution level has failed to generate new resolvents. In fact, reaching this point triggers a theorem prover agent to attempt to import relevant information from other agents as shown in Figure 38. If an agent is successful in importing new knowledge, the resolution processes continues. Otherwise the distributed theorem proving process terminates with no conclusion being drawn about the current theorem.

It is not sufficient in our distributed environment just to wait for local resolution to halt prior to importing new knowledge, since mechanical theorem proving in general may not terminate when dealing with incomplete knowledge. Therefore, as shown in Figure 38, our system must evaluate whether or not progress is being made locally towards a solution. The *Proof Advancing* test is made at the conclusion of each resolution level to heuristically determine if the proof has advanced towards a NIL resolvent. This test cannot give a definitive answer! It can only say "Yes, progress has been made," or that it "does not know." On this basis, local resolution moves to the next level only if the Proof Advancing test concludes that progress has been made. Otherwise, nonlocal knowledge is imported just as if the current level failed to produce new resolvents.

The primary role of this Proof Advancing heuristic is to prevent a distributed agent from expending its resources in an effort that is not likely to be productive. When it is apparent that an agent is not progressing, this heuristic triggers the importation of nonlocal information in order to increase the agent's knowledge with respect to the task at hand, thus potentially permitting the theorem prover to continue in a more productive manner.

Although this heuristic is used to enhance system performance, it should not be made too sensitive. If it appears that an agent is progressing towards a solution, the agent should be allowed to proceed as long as it continues in this direction. Importing knowledge prematurely may be seen to be counterproductive, given the exponential behavior of theorem proving in general.

Definition The *Proof Advancing heuristic* is defined as follows:

1. Given clauses C and D with literal length c and d respectively, a resolvent of these two clauses, R, is said to *make progress* towards a solution if the resolvent clause length, r, is less than $(c + d) - 2$.

2. A proof is also said to have *made progress* whenever a single literal resolvent is generated, or whenever a single literal clause is used to generate a new resolvent.

3. A proof *may not be making progress* whenever (1) and (2) do not occur, and the number of distinct predicate symbols found in the resolvent clauses of two successive resolution levels is not decreasing.

In general, when two clauses are resolved together using binary resolution, the resolvents will always have length less than or equal to the length of the two parent clauses added together, minus the two literals which are consumed by the resolution process. Condition 1 in the Proof Advancement heuristic considers a proof to be advancing whenever a resolvent is generated having length less than its upper bound. This situation happens whenever the substitution used to transform one literal from each parent clause into exact complements of each other (prerequisite for resolution to take place) also generates literals in the parent clauses which are identical. When this happens, identical literals in the parent clauses become one literal in the resolvent clause, thus reducing the resolvent's length to be less than its upper bound.

Condition 2 in the Proof Advancement heuristic definition recognizes that some resolvents are sought after, although their length equals the upper bound. For instance, when a single literal clause C is resolved with clause D of length n, the resolvent has length $c+d-2 = 1+n-2 = n-1$. The importance of these resolvents are recognized by the *Unit Preference* [47] strategy and is discussed in detail later.

Whenever conditions 1 and/or 2 occur during resolution, the proof is considered to be *advancing*. If neither occurs, we do not know if the proof is making progress. Since we do not want our Advancement heuristic to be too sensitive, we do not allow it to kick in and make a knowledge request when conditions 1 or 2 do not occur. Instead, we use a third constraint to ensure that eventually a knowledge request will be made if uncertainty about proof advancement persists. Condition 3 above meets this requirement in that the number of predicate symbols found in the resolvent clauses of two successive resolution levels must be decreasing whenever advancement is not detected. If the number of predicate symbols is not decreasing, a knowledge request is made. On the other hand, when advancement is not detected and the number of predicate symbols is decreasing, eventually this number must be 0. This situation can only occur when the current resolution level fails to generate new resolvents and will be detected by the *New Resolvents* test (refer to Figure 38).

We devote the remainder of this section to explaining how our agents interact, and how an agent determines what knowledge to exchange during the distributed automated reasoning process. We begin by first introducing subsumption, which is a classical reduction strategy used by our knowledge acquisition heuristic. Then the heuristic itself is presented in primitive form. Problems associated with the heuristic are identified, and improvements to the heuristic are made to minimize these problems and increase system performance.

In traditional theorem proving, one technique used to help keep the number of resolvents at a minimum is subsumption.

Definition Clause C_1 *subsumes* clause C_2 if there is a substitution σ such that $C_1\sigma \subseteq C_2$.

As an example of subsumption, suppose $C_1 = Q(x)$ and $C_2 = Q(a) \vee R(b)$. In this case, C_1 subsumes C_2, since the substitution $\sigma = \{a \text{ for } x\}$ satisfies the constraint $C_1\sigma \subseteq C_2$. In other words, since $Q(x)$ is TRUE for all x, then $Q(a)$ must be TRUE. Thus, C_2 is TRUE independent of $R(b)$ and can be reduced to no more than an instance of C_1.

When viewing distributed theorem proving as a distributed problem solving task, we have found subsumption to be an effective and efficient mechanism which plays an important role in the acquisition of nonlocal knowledge by an agent. Subsumption is used by the agent making the request, as well as by the agents responding to the request, in order to minimize the amount of knowledge exchanged among agents.

To demonstrate the role subsumption plays in the exchange of knowledge between agents, consider the following situation in a distributed theorem proving environment. Suppose that local resolution for an agent has been exhausted and nonlocal knowledge must be imported from other agents in order to continue as shown in Figure 38. The first step of the knowledge acquisition process is to determine what knowledge is needed. Is there some way to know what information is lacking locally, and which may be present somewhere else in the system? Are there some requests for knowledge that would be more appropriate to issue than others? Note that these kinds of questions are commonly posed within a group of human experts who are themselves involved in a problem solving task! In an attempt to answer these questions, we will first show how an agent determines what to ask for, then we will show how subsumption is used to formulate better requests and replies.

In our environment, when an agent has reached a point where it is evident that new information must be acquired in order to continue problem solving, the agent formulates a *Priority Set P*. This Priority Set is comprised of information which has been selected as relevant, and if pursued, most likely to advance the agent closer to a solution. Then, based on the contents of P, an appropriate query to other agents is made.

Definition A *Priority Set P* has the form $P = \{C_1, ..., C_n\}$ where each C_i for $0 < i \leq n$ is a clause heuristicly determined to have a high likelihood of furthering the proof towards a NIL resolvent. P is said to have length n, where n is the number of clauses in P.

In distributed theorem proving, the process of determining what to import from other agents can be viewed as the single most important issue contributing to success. In problem solving in general, it is this sense of direction through the problem space that is difficult to achieve by computers, but what is done naturally by humans.

The heuristic we use to determine the *likelihood* of a clause extracts some ideas

found in two popular resolution strategies: *Set of Support* [48] and *Unit Preference* [47]. However, our heuristic is far more than just a combination of these two strategies, as will become evident as we develop our technique. In the Set of Support strategy, resolution between clauses is allowed only if one or both of the clauses have an ancestor which is either the negated theorem itself, or is a resolvent derived from the negated theorem. This technique has proven to be extremely effective in domains with a large knowledge base, and is complete providing that the axiom set is consistent[6]. The incorporation of elements of this strategy into our knowledge acquisition process allows us to focus on the proof, and not on other local knowledge that may have no relevancy to the proof. The Unit Preference strategy is based on the principle that whenever a clause comprised of a single literal is resolved with another clause of length n, the resolvent will always be of length $n-1$. This reduction in resolvent clause length can be viewed as furthering the proof towards a NIL resolvent, which is of length 0. The perception is that a reduction in clause length is seen as furthering the proof towards a solution is also shared by our heuristic. Clauses with fewer literals are regarded as being closer to a solution than clauses of greater length.

Our heuristic determines a *likelihood* that a clause will be relevant in furthering a proof towards a NIL resolvent. This heuristic is based on clause length and clause ancestry. Clauses whose ancestry do not lead back to the negated theorem have no *likelihood* and are assigned the value of 0. Clauses having an ancestry link to the negated theorem have a *likelihood* whose value is the reciprocal of the clause length. Single literal clauses with a negated theorem ancestry have maximum *likelihood* of 1.

As a first cut in distributed theorem proving, one could simply form the Priority Set P using all clauses possessing maximum likelihood. Then P could be sent to all agents, with each agent being requested to return any clause that can resolve with one or more members in P. Unfortunately, P could potentially be large, thus requiring significant processing on behalf of each agent receiving the request. A better strategy would be to first remove any clause in P which is subsumed by another clause in P, as any reduction in the size of P reduces the overhead other agents incur while processing the request.

Though use of subsumption in this way reduces the size of P, it still has the potential to be relatively large. This is undesirable in cases involving a large number of agents and concurrent theorem proving tasks. An alternative approach makes use of a Minimal Literal Set L_{min} derived from P that is defined as follows:

Definition Let each clause C_i in a Priority Set P of length n be of the form $C_i = \{L_1, ..., L_{im_i}\}$ where L_{ij} is a literal, and:

[6]Note in future work we plan to explore distributed theorem proving in domains possessing inconsistent knowledge bases. Under these conditions, the completeness of the Set of Support strategy would be affected, as will most other reduction strategies.

1. $1 \leq i \leq n$;
2. $m_i > 0$ and is the number of literals in clause i;
3. $1 \leq j \leq m_i$.

Then the *Priority Literal Set* L is defined to be the union of literals found in clauses $C_1, ..., C_n$ and has the form $L = \{C_1 \cup ... \cup C_n\}$.

Definition Given L, the Priority Literal Set for $P = \{C_1, ..., C_n\}$, we define L_{min}, the *Minimum Priority Literal Set* for P as follows:

$L_{min} = L - L'$, where $L' = \{L_{jk} \in L \mid$ there is a literal L_{pq} in L, such that L_{jk} is subsumed by $L_{pq}\}$.

After computing the Minimal Priority Literal Set L_{min} from the Priority Set P, the agent transmits L_{min} to other agents and requests knowledge about clauses they may have that resolve with one or more literals in L_{min}. The significance of transmitting L_{min} as opposed to P lies in the savings of computational effort required to process this request. For L_{min} the process involves trying to resolve each literal in L_{min}, one at a time, with each axiom and resolvent known to the agent. For P, resolution would also be attempted with each axiom and resolvent, but would have to be done for each clause in P. The difference between these two methods is that in the former, unification is always done involving a single literal. In the latter, the exponential nature of the unification process becomes evident, because clauses in P do not generally have unit length. In either case, when unification is possible the current clause is tagged to be considered later as part of the reply. Our experimental results indicate that the request process time is dramatically reduced using the L_{min} method, thus enhancing our architecture's performance as discussed in Section 4.5.3.2.

The significance of tagging potential clauses during the unification process is twofold. Once a clause is tagged, it is never again considered when subsequent requests are made by the same agent with respect to the current theorem under investigation. Secondly, subsumption will be used among the tagged clauses to minimize what is returned to the requesting agent. Clearly, this tagging mechanism helps avoid redundancy in what is returned in response to subsequent requests. In addition, tagging can be viewed as an aggregation of knowledge about other agents' activities (Not unlike the behavior evident in the scientific community metaphor [28, 20, 44, 32]). We hope to eventually use this knowledge to gain insight to when newly generated forms of local knowledge should be sent to other agents based on their previous requests.

4.5.3.4 Multiple Agents Example As an example of theorem proving in a multi-agent domain we consider the previous single agent example now distributed

over three agents. The intent of this example is twofold: (1) To demonstrate the ability of an agent to acquire pertinent nonlocal knowledge from other agents as needed; (2) To show how the strategies discussed in the previous section are used in this knowledge acquisition process.

In the single agent example there were six axioms and one negated theorem used in the proof. To emulate a worst case scenario for this problem, we distribute two of the six axioms to each agent, so that no two agents share a common axiom. A copy of the negated theorem is also distributed to each agent, since they are all working to prove the theorem locally. The following clauses show the initial distribution of knowledge among the agents:

Agent A's Initial Knowledge:

A1:	$\neg R(x) \vee S(x, f(x)) \vee T(x)$	(axiom)
A2:	$Q(f(x)) \vee \neg R(x) \vee T(x)$	(axiom)
A3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)

Agent B's Initial Knowledge:

B1:	$P(a)$	(axiom)
B2:	$R(a)$	(axiom)
B3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)

Agent C's Initial Knowledge:

C1:	$P(y) \vee \neg S(a, y)$	(axiom)
C2:	$\neg P(x) \vee \neg T(x)$	(axiom)
C3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)

Although all agents generate the proof concurrently, we only focus on the solution from agent A's perspective in this example. Figure 39 shows the resolution tree used by agent A in the proof and Figure 40 is a listing of all clauses appearing in Figure 39.

Referring to Figure 39, it can be seen that resolvent A4 is generated directly from local clauses A2 and A3. However, A4 is the only resolvent possible in agent A's first level of resolution. Since no resolvents are possible during the second level of resolution, agent A is forced to request information from other agents in order to continue.

The first step in determining what to request from other agents is to formulate Priority Set P. This is done by considering only those clauses with an ancestry related back to the negated clause, and then extracting the clauses with the highest likelihood of furthering the proof towards a NIL resolvent. As discussed in Section 4.5.3.3 our

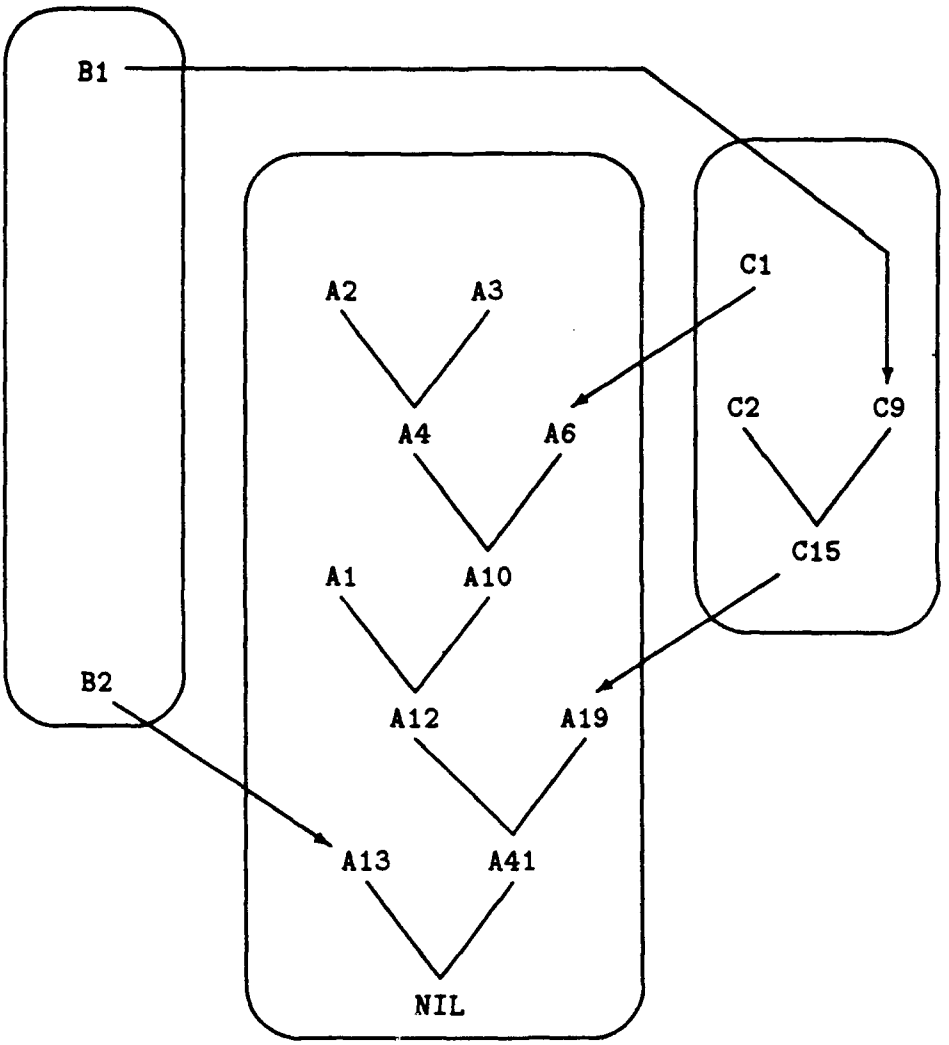


Figure 39: Proof of Example Theorem in a Multi-Agent Domain

Agent A's Axioms, Negated Theorem, and Pertinent Resolvents:

A1:	$\neg R(x) \vee S(x, f(x)) \vee T(x)$	(axiom)
A2:	$Q(f(x)) \vee \neg R(x) \vee T(x)$	(axiom)
A3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)
A4:	$\neg P(f(x)) \vee \neg R(x) \vee T(x)$	(resolvent of A2 and A3)
A6:	$P(y) \vee \neg S(a, y)$	(imported C1)
A10:	$\neg R(x1) \vee \neg S(a, f(x1)) \vee T(x1)$	(resolvent of A4 and A6)
A12:	$\neg R(a) \vee T(a)$	(resolvent of A1 and A10)
A13:	$R(a)$	(imported B2)
A19:	$\neg T(a)$	(imported C15)
A41:	$\neg R(a)$	(resolvent of A12 and A19)
A55:	NIL	(resolvent of A13 and A41)

Agent B's Axioms, Negated Theorem, and Pertinent Resolvents:

B1:	$P(a)$	(axiom)
B2:	$R(a)$	(axiom)
B3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)

Agent C's Axioms, Negated Theorem, and Pertinent Resolvents:

C1:	$P(y) \vee \neg S(a, y)$	(axiom)
C2:	$\neg P(x) \vee \neg T(x)$	(axiom)
C3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)
C9:	$P(a)$	(imported B1)
C15:	$\neg T(a)$	(resolvent of C2 and C9)

Figure 40: Solution Space Clause List for Multi-Agent Example

heuristic for determining *likelihood* is the reciprocal of clause length, so that single literal clauses have highest likelihood (1).

At this point in the proof procedure there are only four clauses known to agent A:

A1:	$\neg R(x) \vee S(x, f(x)) \vee T(x)$	(axiom)
A2:	$Q(f(x)) \vee \neg R(x) \vee T(x)$	(axiom)
A3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)
A4:	$\neg P(f(x)) \vee \neg R(x) \vee T(x)$	(resolvent of A2 and A3)

We see that clauses A1 and A2 have no negated theorem ancestry, thus their *likelihood* is 0. A3 is the negated theorem and has likelihood of 0.5, since its clause length is 2. The likelihood of A4 is 0.33 based on its clause length of 3. Therefore, agent A's current Priority Set P is:

$$P = \{A3\} = \{\ \{\neg P(x) \vee \neg Q(x)\}\ \} = \{\ \{\neg P(x), \neg Q(x)\}\ \}^7$$

Given P, the Priority Literal Set L is defined to be the union of all literals found in each clause of P. Furthermore, L_{min} is a subset of L, where any literal in L subsumed by any other literal in L has been removed. Therefore, for this example we see that:

$$L_{min} = L = \{\neg P(x), \neg Q(x)\}$$

At this point agent A now has a Minimum Literal Set. This set contains literals from clauses which have been rated as having the highest likelihood of furthering the proof towards a NIL resolvent. In order to further the proof towards a NIL resolvent, new clauses that can resolve with these literals must be imported from other agents. In other words, these imported clauses must contain at least one literal which can resolve with a literal in L_{min}. Thus these imported clauses must contain at least one literal which can be unified with the complement of a literal in L_{min}.

Definition Given a Minimum Priority Literal Set $L_{min} = \{Q_1, ..., Q_n\}$ of length n, where each Q_i is a literal for $0 < i \leq n$, then the *Minimum Priority Negated Literal Set* NL_{min} has the form $NL_{min} = \{R_1, ..., R_n\}$, where each $R_i = \neg Q_i$ for $0 < i \leq n$.

[7]Recall that in theorem proving, knowledge is represented in conjunctive normal form, where each conjunct is either an axiom or resolvent. In this set notation, the AND connective is dropped and a comma is used to separate clauses. Furthermore, each clause is in disjunctive normal form. Again, in set notation, the OR connective is dropped and a comma is used to separate literals.

In DARES, the Minimum Priority Negated Literal Set NL_{min} is transmitted to the other agents, and these agents are asked to return clauses that can be unified with at least one literal of NL_{min}. Although we could have transmitted L_{min} rather than NL_{min}, use of NL_{min} allows the responding agent to perform unification directly on the request, thus minimizing the response time (see Section 4.5.3.2).

There is one more constraint used by an agent when importing information from other agents. The literal length of any clause returned must be no greater than the length of the clauses in priority set P. This added restriction is included in the request, and is based upon the heuristic that clauses of lower literal length have higher *likelihoods*. Initial experimental results strongly support the use of this length limitation based on priority set clause length.

Returning to our example, we see that agent A's request is for clauses of length 2 or less which can unify with at least on literal from NL_{min}, where

$$NL_{min} = \{P(x), Q(x)\}$$

In response to this request, the following two clauses are imported from agents B and C respectively:

A5:	$P(a)$	(imported B1)
A6:	$P(y) \vee \neg S(a, y)$	(imported C1)

Figure 39 shows that imported clause C1 becomes local clause A6 and is resolved with A4 to yield clause A10. It also shows that A1 and A10 are resolved together to form A12. However, once reaching this point, local resolution is again exhausted and cannot proceed without importing new nonlocal knowledge.

This time the acquisition process for new knowledge is based upon the following twelve local clauses, two of which are single literals:

A1:	$\neg R(x) \vee S(x, f(x)) \vee T(x)$	(axiom)
A2:	$Q(f(x)) \vee \neg R(x) \vee T(x)$	(axiom)
A3:	$\neg P(x) \vee \neg Q(x)$	(negated theorem)
A4:	$\neg P(f(x)) \vee \neg R(x) \vee T(x)$	(resolvent of A2 and A3)
A5:	$P(a)$	(imported B1)
A6:	$P(y) \vee \neg S(a, y)$	(imported C1)
A7:	$P(f(a)) \vee \neg R(a) \vee T(a)$	(resolvent of A1 and A6)
A8:	$\neg Q(a)$	(resolvent of A3 and A5)

A9: $\neg Q(x1) \vee \neg S(a, x1)$ (resolvent of A3 and A6)

A10: $\neg R(x1) \vee \neg S(a, f(x1)) \vee T(x1)$ (resolvent of A4 and A6)

A11: $\neg Q(f(a)) \vee \neg R(a) \vee T(a)$ (resolvent of A1 and A9)

A12: $\neg R(a) \vee T(a)$ (resolvent of A1 and A10)

In its first attempt to import new knowledge based on its current clause set, agent A uses the single literal clauses A5 and A8 to form its current Priority Set P_1, Priority Literal Set L_1, Minimum Priority Literal Set L_{min_1}, and the Minimum Priority Negated Literal Set NL_{min_1} as follows:

$$P_1 = \{A5, A8\} = \{\ \{P(a)\},\ \{\neg Q(a)\}\ \}$$

$$L_1 = \{P(a), \neg Q(a)\}$$

$$L_{min_1} = \{P(a), \neg Q(a)\}$$

$$NL_{min_1} = \{\neg P(a), Q(a)\}$$

Sending NL_{min} to agents B and C fails to import any knowledge. Now agent A must relax its likelihood constraint and try again. This time clauses of length 2 or less are used to derive the following:

$$\begin{aligned} P_2 &= \{A3, A5, A6, A8, A9, A12\} \\ &= \{\ \{\neg P(x), \neg Q(x)\},\ \{P(a)\},\ \{P(y), \neg S(a, y)\},\ \{\neg Q(a)\}, \\ &\quad \{\neg Q(x), \neg S(a, x)\},\ \{\neg R(a), T(a)\}\ \} \end{aligned}$$

$$L_2 = \{\neg P(x), \neg Q(x), P(a), P(y), \neg S(a, y), \neg Q(a), \neg Q(x), \neg S(a, x), \neg R(a), T(a)\}$$

$$L_{min_2} = \{\neg P(x), \neg Q(x), P(y), \neg S(a, x), \neg R(a), T(a)\}$$

$$NL_{min_2} = \{P(x), Q(x), \neg P(y), S(a, x), R(a), \neg T(a)\}$$

Note that clauses A5 and A8, which were the basis for the previous request, are also covered by this current request. This ensures that if new knowledge pertinent to the previous request has since been derived or acquired by agents B and C, it will be returned in response to this current request. We would also like to point out that the tagging mechanism described in Section 4.5.3.2 does not allow clauses previously considered to be tested again. This reduces the overhead associated with including A5 and A8 in P_2. However, as our system matures, we envision that the inclusion of A5 and A8 will not be necessary, since agents will be able to realize when to automatically forward new information that is relevant to previous requests.

In response to what agents B and C return based on this current level 2 query, the following seven clauses are instantiated in agent A's environment:

A13:	$R(a)$	(imported B2)
A14:	$Q(f(a)) \vee \neg R(a)$	(imported B9)
A15:	$Q(f(a)) \vee T(a)$	(imported B10)
A16:	$P(f(a)) \vee T(a)$	(imported B11)
A17:	$\neg P(a) \vee Q(f(a))$	(imported B12)
A18:	$\neg P(x) \vee \neg T(x)$	(imported C2)
A19:	$\neg T(a)$	(imported C15)

This new imported information now allows agent A to continue the resolution process. We see in Figure 39 that resolvent A12 now resolves with A19 (imported from agent C) to form A41. This new resolvent along with A13 (imported from agent B) are then used during the next level of resolution to derive a NIL resolvent. This NIL resolvent indicates that there is a direct contradiction between the axioms and the assumption that the negated theorem is TRUE. Therefore, the negated theorem must be FALSE. Thus, the theorem is shown by refutation to logically follow from the axioms.

4.5.3.5 Status In the previous example, agent A is able to complete the proof in about half the time required by a single agent possessing complete knowledge of the axioms. Furthermore, for the multi-agent case where all three agents solve the theorem completely, the average number of resolvents generated per agent is 37 as opposed to 69 in the single agent domain. This average number of resolvents per agent is even less when all efforts to prove the theorem are abandoned as soon as one of the distributed agents completes the task.

We are finding that these performance characteristics are far better for proofs that require a fair amount of computational effort. For example, Figure 41 is the time characteristics recorded by DARES for a problem comprised of six axioms and two negated theorem clauses. You will note that we have taken the natural log of the time, since there is an order of magnitude enhancement in the performance. The time axis has also been normalized so that the single agent case has a value of unity. The Redundancy axis has been scaled by a factor of five ($0 \leq R \leq 1$) for legibility.

We are observing results indicating that when there is a high level of redundancy among the agent's local knowledge, that DARES' run time begins to approach that of the single agent. This can be attributed to a very low interaction rate between agents, since each agent's theorem prover has enough local knowledge to advance the proof almost to completion by itself. However, when the proof does stop advancing and the agent is forced to import nonlocal knowledge to continue, the request made is usually very specific and available. In other words, our knowledge acquisition heuristic for agents possessing a high redundancy rate successfully identifies what portions of the

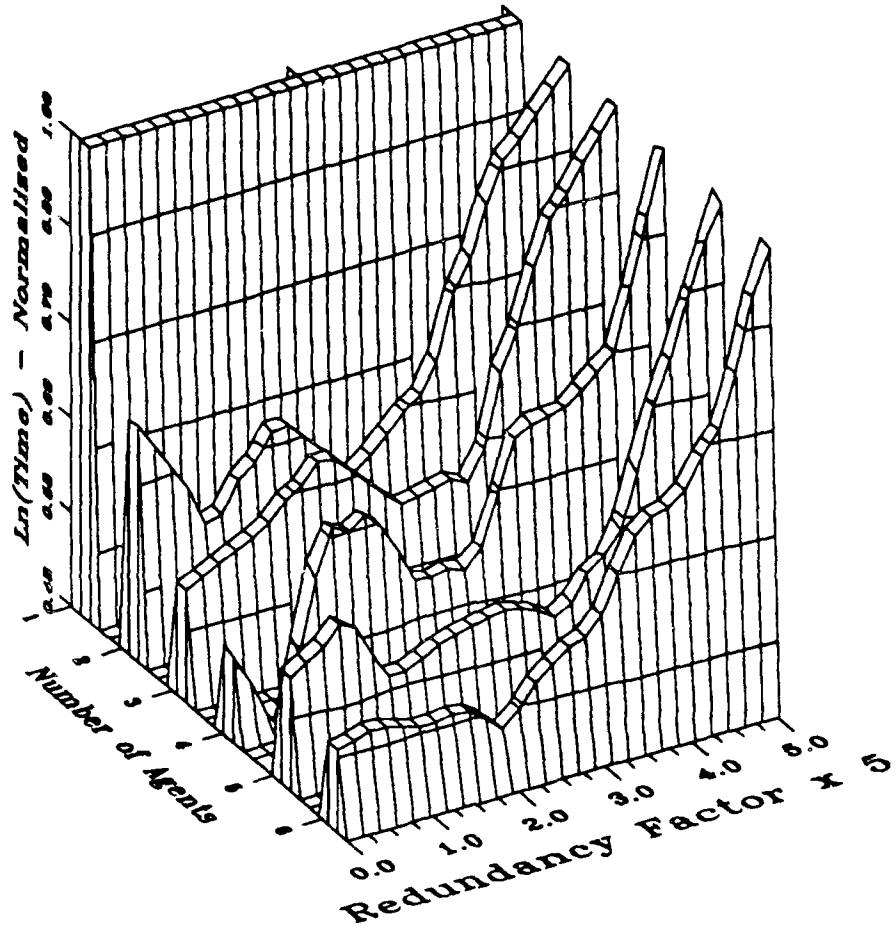


Figure 41: Typical Time Characteristics for DARES.

proof are close to a solution and is able to make very specific requests based on this assessment. Furthermore, since other agents also have high redundancy rates and have had ample time to advance their local efforts, the information being requested by the first agent is typically available. This behavior is evident in Figure 41.

Figure 41 also suggests that performing distributed theorem proving is best done by many agents possessing little redundancy. For this circumstance, each agent can be thought of as becoming specialized. At the start of the distributed theorem proving process, each agent advances its part of the proof as far as it can before making a knowledge request. At the time such a request is made, the agent has begun to concentrate its efforts on its local advancements and imports knowledge relative to this acquired focus. Since redundancy is low, we see the agents becoming specialized in different areas, which reduces search space overlap between agents and leads to increased system performance.

Another interesting characteristic depicted in Figure 41 is a small mound that is prevalent in data taken for networks comprised of small numbers of agents. It appears that this mound shifts incrementally out of the picture as the number of agents increases. One explanation for this phenomenon that is currently under investigation has to do with the selection of the Priority Set by our knowledge importation heuristic. At the time resolution first halts for an agent in the area under this mound, it may be that the proof locally has not advanced far enough for the heuristic to generate a meaningful Priority Set. Instead, the Priority Set may be very general, thus leading to an influx of information which is also general in nature. A second factor contributing to this glitch may be due to replies to knowledge requests that are themselves too general. This could occur when an agent formulates a reply based on its local theorem prover state, which may not have progressed very far.

It appears that there is a natural pruning of the search space that occurs when knowledge is distributed throughout the network. This can be attributed to the difference between traditional theorem proving and our distributed approach. When a single agent performs theorem proving, resolvents are mechanically produced based on all information at hand. In our distributed environment, this mechanical process terminates relatively quickly when compared to the single agent case due to the lack of local knowledge. Instead of importing everything possible from other agents (which would emulate the single agent situation) we use the heuristic of attempting to acquire knowledge that is most likely to advance the proof towards a solution. If we fail to advance, we then import information rated less likely to be useful. It is the use of this heuristic that results in a pruning of the search space. This heuristic also guarantees completeness of the distributed theorem algorithm, since every clause that can be used in resolution will be acquired if necessary.

Currently, we are still collecting data for more complex problems. However, these new experiments are also yielding results similar to those mentioned above. We are finding that as the complexity of our test problems increase, so does the performance of our distributed system. We are continuing work to develop other strategies for enhancing the capabilities and performance enhancement of this particular distributed problem solving environment based on an agent's local knowledge about the current state of the domain, previous interactions with other agents, and knowledge about theorem proving in general.

4.6 Development of an AI Infrastructure

One of the goals in forming the Northeast AI Consortium was to foster the growth of AI research activities at the member universities. While some members of the NAIC brought a distinguished record of past AI research to this effort, other members, such as Clarkson University, had not previously been active in AI research. In order to

initiate a long-term, viable research program of high quality, a substantial investment in building up the basic infrastructure was required. This section briefly documents the steps taken at Clarkson in making this investment.

4.6.1 Development of AI Research at Clarkson University

The two most important ingredients of a successful research program are first-rate facilities and high quality personnel. Using internal research funding (with some assistance from the CASE center at Syracuse University), Clarkson began the development of an AI research lab with the purchase of a Symbolics 3670 LISP machine. Approximately two years later, the lab was expanded through the joint sponsorship of Clarkson and a Department of Defense Research Equipment grant. In this last year the lab has expanded further as a result of the generosity of Texas Instruments, Inc. The AI Laboratory now has five LISP machines and a laser printer; it is networked with the campus Sun file servers and the external research community over ARPAnet.

Research personnel have grown from the original two faculty, and no graduate students, to five or six faculty with approximately six to eight graduate students. We have added three graduate level courses in AI, introduced a required undergraduate course in symbolic computation based on the LISP dialect, Scheme, and have modified a database management course to include introductory material on knowledge base management. These curricular changes are especially important in ensuring that future graduate students who come into the program will have an appropriate academic background for AI research.

The most visible measure of research activities from an external viewpoint is, of course, scholarly publications in the form of theses, journal articles, books and conference papers. The bibliography for this report includes the citations to most of our publications. Our work has been selected to appear in two different collections of papers on DAI, was solicited for one chapter in a book, and most recently received a citation for the best paper at the Ninth Workshop on Distributed Artificial Intelligence.

4.6.2 Coordination within the NAIC

From the initial meetings of the NAIC, personnel from Clarkson University have expressed interest in working cooperatively with other researchers within the NAIC. There have been two excellent examples of the realization of this interest.

First, it was the efforts of Dr. Susan Conry at Clarkson, working closely with the principal investigators at each of the other NAIC members, which resulted in the successful proposal to the Department of Defense for LISP machines at each university. The most significant aspect of this proposal was not that it provided

additional computational facilities, but that it ensured a single common environment available to all researchers within the NAIC. Having such an environment has clearly encouraged the various joint research projects which have developed among the NAIC members subsequently.

The second example of cooperative efforts was a seven month sabbatical by Drs. Conry and Meyer spent working with Dr. Victor Lesser at the University of Massachusetts. During this period one of Clarkson's graduate students also studied in residence at the University of Massachusetts. As a result of this joint work, we have written joint papers and have continued our association with Dr. Lesser on new projects.

4.6.3 Technology Transfer to other RADC efforts

Another measure of the growth of AI research at Clarkson is the increased level of interaction with other research and development teams involved in similar activities. An important goal of Clarkson's research effort in this project has been one of doing work which is relevant to the needs of RADC. As a secondary objective to support this goal, we have sought to establish close working relationships with other RADC contractors, especially those engaged in the development of applications which might involve the use of AI technology arising from our research. There are two principal actions we have taken to further these objectives.

First, we have participated as an active observer in several other RADC communications system control projects in order to enhance our understanding of the overall problem area. For example, near the beginning of this project we were involved with the Integrated DCS Control Study performed by the Honeywell Corp. for RADC. Our participation included a ten day field survey (along with personnel from RADC, Honeywell Corp., and the Air Force Communications Command) of several DCS sites in Europe. The knowledge gained from this experience has been especially valuable in our development of a model of system control for a DCS-like network. A second example is found in our participation with Lincoln Laboratories as a consultant in the development of the expert tech controller project. Through this interaction we have gained valuable insight into the nature of the problems faced by human operators and the expert knowledge they used in solving these problems. More recently, we have participated in a project at RADC to enhance the multinet gateway. In this effort we have broadened our understanding of network management and control to include tactical communication networks as well as the DCS.

The second step in making the results of our research relevant to the needs of RADC has been to assist in the transfer of this technology from the university laboratory environment to the industrial development process. For example, in the case of Lincoln Laboratories, we have continued to maintain a close working relationship

so that we were aware of the problems they were addressing, and similarly they were aware of our work. The specific benefits have been that the two efforts have been complementary rather than duplicative. We have provided early copies of our software systems so that they could understand what we had accomplished. In turn, they have invited us to visit them and participate in detailed technical discussions about their work. This level of cooperation has proven to be of great value in assisting technology transfer. We have also worked with Harris Corp. as a subcontractor in developing a methodology for the design of AI applications for communications network management in the SDI (Space Defense Initiative) area. This type activity represents a direct transfer of expertise developed within the university research labs to an industrial development project.

4.7 Future Developments

At this point in the report we assess the results and accomplishments achieved thus far and look forward to areas for future development. As described in the previous sections we have designed and implemented a testbed environment, known as DAISY, in which distributed artificial intelligence (DAI) issues may be studied. We have used DAISY to demonstrate the applicability of DAI techniques for communications network management and control. However, there are several outstanding issues and questions which remain unanswered at this time. We find the areas which need improvement and further study are as follows: (1) improvements in DAISY, (2) generalization and standardization of domain knowledge represented, (3) enhancements to MATMS and extension of truth maintenance to distributed knowledge bases, (4) demonstration of cooperative problem solving among a group of local performance assessment agents using DARES, and (5) investigation of the performance of our current problem solving strategies under network dynamics and consideration of alternative designs.

4.7.1 Improvements in DAISY

DAISY is the basic testbed for development and testing of prototype DAI paradigms. In large part, DAISY is domain-independent; it could be used for developing DAI problem solvers for any particular problem area. DAISY does include some components which have been designed especially for the problems in communications network management. For example, GUS was developed to provide a graphical interface for the knowledge representation modules, and thus it is specific to communication networks and in particular to the network model we used. Having used the overall system during the last few years, we have found graphics to be essential not only in the original knowledge representation step, but also in displaying the progress of distributed problem solvers in cooperatively finding a solution. We believe the basic

graphics capability of GUS should be made independent of the problem domain and included as part of the domain-independent part of DAISY. This would provide a common graphics framework for any DAI module to use in displaying on-line graphic representations of the problem state.

4.7.2 Enhancements in Domain Knowledge Representation

There are two limitations in our current approach to domain knowledge representation. First, we do not provide an extendable language for network representation; the choice of network objects, such as link type, radio type, and switch type, are predefined and may not be augmented except by changes in program code. We recommend changes be made in the knowledge representation scheme which would allow the user to extend object definitions to incorporate new types not originally programmed. In conjunction with this, we would suggest that the basic network model be broadened to incorporate more generic network types including packet switched networks, integrated services digital networks (ISDN), and networks with both fixed and mobile nodes.

The second limitation of our current approach is the use of a non-standard object-oriented database for basic network configuration knowledge. While wishing to continue the benefits of the object-oriented approach, we have recognized the importance of utilizing a more standardized method. The compromise we suggest is to investigate the use of a conventional SQL-based relational database for the largest portion of network configuration data, coupled with an object-oriented interface which would include the graphical interface based on the generic graphics modules as discussed in the previous section. This approach could be viewed as providing an intelligent front-end to a more conventional relational database. We believe an investigation of this area would be of interest in a number of AI problem domains.

4.7.3 Enhancements and Extensions to MATMS

There are four primary issues for future investigations involving the multi-agent truth maintenance system (MATMS): (1) creating a third type of belief, (2) grading assumptions and inferences with certainty factors, (3) performing distributed truth maintenance, and (4) studying techniques for resolving inconsistency across problem solvers.

In Section 4.5.2.7.4, Example 2 illustrates the possible need for a third type of belief. In that example, PA had reasoned that a trunk ($trkx$) was down because multiple circuits on the trunk failed at the same time. However, since PA had no direct evidence of the trunk failure, it was taken as a non-default assumption rather than an inference. This belief does not cleanly fit into either category.

In analyzing the example if "*trkx* is down" were called an inference and the scenario continued, problems would arise later in deciding how to retract it. In general, an inference is only retracted when an assumption upon which it is based is retracted. In this case none of the underlying assumptions (about the failure of multiple circuits at the same time) should be retracted. Yet, FI must eventually retract either the assumption that *trkx* tests good or the belief that *trkx* is down.

The belief "*trkx* is down" is not strictly an assumption because its validity is dependent upon other beliefs. For example, if "*ckt*1 is down" were retracted, then "*trkx* is down" would also be retracted. The problem in treating "*trkx* is down" as an assumption is that the MATMS will not automatically retract it when an agent removes "*ckt*1 is down" from its belief set. The current design insists that the problem solver explicitly retract each belief in this case. Therefore, clearly part of the the MATMS's purpose is defeated when "*trkx* is down" is treated as an assumption.

Overall, the problem is that there will be beliefs which are difficult to categorize. A third category of beliefs, perhaps called *reasoned assumptions*, should be pursued. Reasoned assumptions should be handled like assumptions in some ways, and like inferences in other ways.

Another issue which requires further study involves the determination of which assumptions should be removed when inconsistencies arise. When a problem solver's belief set becomes inconsistent, it must usually remove one or more non-default assumptions in order to correct the problem. The decision as to which assumption(s) should be removed is very difficult to make. For example, if the MATMS provides a problem solver with the knowledge that two assumptions in its belief set, A_1 and A_2, are inconsistent, how does the problem solver decide which assumption to remove?

At this time, assumptions provide a purely black-and-white world; either they are present within a problem solver's belief set, or they are not. The reasons why a particular non-default assumption is made is kept solely within each problem solver making the assumption. For example, as discussed above, PA used the rule "If multiple circuits on a trunk fail at the same time, assume that the trunk has failed" to conclude that "*trkx* is down." Clearly PA knows why it made the assumption; however, this knowledge is not kept within the MATMS. The MATMS is designed only to keep track of which inferences depend upon the assumption, *not why the assumption was originally made by the problem solver*. There is no real means by which the problem solver can compare assumptions, except by chronological backtracking to determine *why it made the assumption*. Even then, the choice may not be evident.

An alternative design would suggest keeping the reason why an assumption was made with the assumption itself in the MATMS knowledge base. The MATMS could then attempt to resolve inconsistencies. One approach would be to include a certainty factor with each reason. The assumption with the "lower degree of certainty" would be rejected. Of course this creates another problem, precisely how to assign certainty

factors to assumptions. It is exceedingly difficult to place a single measure upon an assumption in order to allow comparisons between assumptions. It is much easier to simply compare a set of assumptions as the inconsistencies arise to determine which to discard. If certainty factors could be reasonably assigned to assumptions, then choosing between two viable default assumptions, an action which is almost impossible now, could be resolved.

Distributed truth maintenance has been a long term concern in working with distributed problem solving systems. Although not particularly evident (and not presented in this work because distributed truth maintenance is not the immediate purpose of the MATMS), the MATMS incorporates the basic framework for distributed truth maintenance.

Distributed truth maintenance is necessary for any distributed knowledge base in which at least some of the knowledge is replicated. In our domain, trunk and circuit information is replicated in certain knowledge bases. When an agent in one subregion determines that a particular trunk is down and enters this knowledge into its MATMS, then the KBM of the subregion should inform all other KBMs with knowledge of the trunk that it believes that the trunk is down. Mason and Johnson describe the fundamentals of distributed truth maintenance in [36].

In much the same way that the KBM must seek to resolve inconsistency among the problem solvers in its local system, the KBM must also attempt to resolve inconsistency between itself and other KBMs. In both situations we are concerned with questions of a problem solver's scope of knowledge, and credibility. An interesting area we have identified for future investigation is the detection of rational *vs.* irrational behavior by a problem solving agent. Distributed truth maintenance is especially interesting because it necessitates distributed control. A single KBM resolving conflict within a single knowledge base implies central control. When KBM agents and the knowledge are distributed, a cooperative control strategy is required. These issues clearly merit further study.

4.7.4 Cooperative Assessment of Network State (CANS)

As discussed earlier, DARES is a general distributed reasoning system with no domain knowledge. In order to demonstrate cooperative performance assessment, local agents with detailed knowledge about communication network performance characteristics must be designed, tested, and integrated. The combined system of these local, rule-based PA agents cooperating using DARES is known as CANS. At the time of this report, CANS is currently being implemented. We believe DARES has demonstrated the feasibility of distributed reasoning, so the primary remaining issue here is the ability to design a clean interface between the local, domain-knowledgeable agent and the cooperation paradigm represented by DARES.

4.7.5 Consideration of Dynamic Environments

A significant research problem for many application areas of AI is the ability to operate under dynamic contexts. The classical approach to knowledge-based expert systems has generally assumed a static, off-line operating scenario as compared with the dynamic, on-line, near real time situation demanded by most real-world problems. Certainly a complete network management and control system is characterized by its abilty to perform in highly dynamic situations which arise.

There are a number of important areas in our own work which require additional investigation and testing to determine their performance under dynamic network conditions. The work on distributed planning for service restoral and distributed situation assessment assumes that we are solving problems given a "snapshot" of the state of the network. Our negotiation protocols that determine a set of desired restoral actions, for example, assume that all of the assets that are used in restoral are available when the control actions are to be executed. It is clear that this assumption is not entirely realistic.

First, careful and comprehensive testing is required under a set of dynamic scenarios. These tests would reveal the areas of weakness and strength in each of the problem solvers for various network dynamics. Specific problem solving strategies could then be identified for further development.

Extensions and enhancements to these problem solving strategies should be investigated. These enhancements would address such issues as: (1) adapting the strategy used to existing network conditions, (2) incremental problem solving as a mechanism for responding to dynamic system changes in a timely fashion, (3) problem solving at different levels of abstraction, and (4) regional variation in problem solving strategy. The previously discussed extension to more generic network models which include mobile elements would provide an excellent basis for testing these new ideas.

4.8 Conclusion

This report has described the important achievements of this research project during the past five years. There are clearly many areas not yet fully investigated in this work, and there have been several new issues raised in the course of this study. Rather than this report serving as a conclusion of this work, it has set the stage for new work in investigating the design of distributed AI systems. These systems have application not only in communications network management, but also in other areas of command, control, communication, and intelligence information processing.

References

[1] N. Aiello. User directed control of parallelism; the CAGE system. In *Proceedings of the Expert Systems Workshop*, pages 146–151, Asilomar, Pacific Grove, CA, April 1986.

[2] P. E. Allen, S. Bose, E. M. Clarke, and S. Michaylov. PARTHENON: A parallel theorem prover for non-horn clauses. In *9th International Conference on Automated Deduction*, pages 764–765. Springer-Verlag, May 1988.

[3] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume I. William Kaufmann, Inc., Los Altos, California, 1981.

[4] D. G. Bobrow. Qualitative reasoning about physical systems: An introduction. *Artificial Intelligence*, 24(1-3):1–5, December 1984.

[5] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[6] Marshall P. Cline. *A Fast Parallel Algorithm for N-ary Unification with A.I. Applications.* PhD thesis, Clarkson University, Potsdam, NY 13676, April 1989.

[7] S. E. Conry, R. A. Meyer, and V. R. Lesser. Multistage negotiation in distributed planning. Technical Report COINS Technical Report 86-67, University of Massachusetts at Amherst, Department of Computer and Information Science, Amherst, MA 01003, December 1986.

[8] S. E. Conry, R. A. Meyer, and V. R. Lesser. Multistage negotiation in distributed planning. In A. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence.* Morgan Kaufman Publishers, California, August 1988.

[9] S. E. Conry, R. A. Meyer, and R. P. Pope. Mechanisms for assessing nonlocal impact of local decisions in distributed planning. In M. Huhns and L. Gasser, editors, *Distributed Artificial Intelligence, Volume II.* Pittman Publishing, Ltd. and Morgan Kaufman Publishers, August 1989.

[10] S. E. Conry, R. A. Meyer, and J. E. Searleman. A shared knowledge base for independent problem solving agents. In *IEEE Proceedings of the Expert Systems in Government Symposium*, McLean, Virginia, October 1985. IEEE Computer Society.

[11] B. Davies. CAREL: A visible distributed lisp. In *Proceedings of the Expert Systems Workshop*, pages 171–178, Asilomar, Pacific Grove, CA, April 1986.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, March 1960.

[13] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

[14] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[15] J. de Kleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[16] J. de Kleer, J. Doyle, G. L. Steele Jr., and G. J. Sussman. AMORD: Explicit control of reasoning. *SIGART Newsletter*, pages 116–125.

[17] Defense Communications Agency. *DCA Circular 310-70-1*, March 1984. Draft.

[18] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[19] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, September 1985.

[20] Mark S. Fox. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):70–80, January 1981.

[21] L. Gasser. MACE, a multi-agent computing environment. Technical report, University of Southern California, March 1986. USC-DPS Group.

[22] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[23] C. Hewitt. Concurrency in intelligent systems. *AI Expert (Premier)*, pages 44–50, 1986.

[24] Brian R. Hogencamp. GUS: A graphical user interface for capturing structural knowledge. Master's thesis, Clarkson University, Potsdam, New York, January 1987.

[25] D. J. Mac Intosh, S. E. Conry, and R. A. Meyer. Role of knowledge in distributed problem solving. In *9th Annual Distributed Artificial Intelligence Workshop*. American Association for Artificial Intelligence (AAAI), September 1989.

[26] Douglas J. Mac Intosh and Susan E. Conry. A distributed development environment for distributed expert systems. In *Proceedings of the Third Annual Expert Systems in Government Conference*, pages 19–23, Washington, D.C., October 1987. Computer Society Press of the IEEE. (also available as NAIC Technical Report TR-8714).

[27] Douglas J. Mac Intosh and Susan E. Conry. SIMULACT: A generic tool for simulating distributed systems. In *Proceedings of the Eastern Simulation Conference*, pages 18–23, Orlando, Florida, April 1987. The Society for Computer Simulation. (also available as NAIC Technical Report TR-8713).

[28] William A. Kornfeld and Carl E. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):24–33, January 1981.

[29] B. Kuipers. Commonsense reasoning about causality: Deriving behavior from structure. *Artificial Intelligence*, 24(1-3):169–203, December 1984.

[30] Kazuhiro Kuwabara and Victor R. Lesser. Extended protocol for multistage negotiation. Personal communication.

[31] H. Leiberman. There's more to menu systems than meets the screen. In *ACM/SIGGRAPH*, pages 181–189, San Francisco, CA, July 1985.

[32] Victor R. Lesser and Daniel D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):81–96, January 1981.

[33] Victor R. Lesser and Daniel D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, (3):15–33, Fall 1983.

[34] M. C. Maguire. A review of human factors guidelines and techniques for the design of graphical human-computer interfaces. *Computers and Graphics*, 9(3):221–235, 1985.

[35] J. Martins and S. Shapiro. A model for belief revision. In *Proceedings of the Non-Monotonic Reasoning Workshop*, pages 241–294. AAAI, 1984.

[36] Cindy L. Mason and Rowland R. Johnson. DATMS: A framework for distributed assumption based reasoning. In *Proceedings of the 1988 Workshop on Distributed Artificial Intelligence*, Lake Arrowhead, CA, May 1988.

[37] D. McAllester. A three-values truth maintenance system. Technical Report Memo 473, MIT AI Lab, 1978.

[38] R. A. Meyer and S. E. Conry. Communications. In Thomas C. Bartee, editor, *Expert Systems and Artificial Intelligence: Applications and Management*, chapter 3, pages 61–96. Howard W. Sams and Co., Indianapolis, 1988.

[39] Randall P. Pope. Role recognition in multiagent distributed planning. Master's thesis, Clarkson University, Potsdam, N. Y., September 1988.

[40] J. Rice. Poligon, a system for parallel problem solving. In *Proceedings of the Expert Systems Workshop*, pages 152–159, Asilomar, Pacific Grove, CA, April 1986.

[41] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, January 1965.

[42] Eric Schoen. The CAOS system. Technical Report KSL-86-22, Stanford University, March 1986.

[43] Reid G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(12), December 1980.

[44] Reid G. Smith and Randall Davis. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1):61–70, January 1981.

[45] Guy L. Steele Jr. *Common LISP.* Digital Press, Burlington, MA, 1984.

[46] S. Vinter, K. Ramamritham, and D. Stemple. Recoverable communicating actions in gutenberg. In *Proceedings of the International Conference on Distributed Computing Systems*, May 1986.

[47] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, Engelwood Cliffs, NJ., 1984.

[48] L. Wos and G. A. Robinson. Paramodulation and set of support. In *Proceedings of the IRIA Symposium on Automatic Demonstration*, pages 267–310. Springer-Verlag, 1968.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.